UNIVERSITY OF OSLO
Department of Informatics

Sikkerhet i rike internett-applikasjoner

Øyvind Mengshoel Reistad

2. november 2009

Sammendrag

Temaet for denne masteroppgaven er sikkerhet i rike internettapplikasjoner. Oppgaven presenterer ulike rammeverk for utvikling av denne typen applikasjoner, og undersøker deretter hvilke sikkerhetsutfordringer man som utvikler må være oppmerksom på når man benytter rammeverk for utvikling av rike internettapplikasjoner. Dette gjøres ved å først studere hvilke sikkerhetsutfordringer som gjelder for tradisjonelle webapplikasjoner. Det vil deretter bli gjennomført sikkerhetstester for å undersøke om noen av disse svakhetene også kan ramme rike internettapplikasjoner. I tillegg vil det bli vurdert om det eventuelt har blitt introdusert nye sikkerhetsutfordringer ved innføringen av denne typen webapplikasjoner.

Innhold

Figurer

FIGURER

Tabeller

TABELLER

Kapittel 1

Innledning

1.1 Innledning

Dette dokumentet er skrevet av Øyvind Mengshoel Reistad og er en masteroppgave om sikkerhet i rike internettapplikasjoner (forkortet RIA). Oppgaven er skrevet under veiledning av Thomas Johan Eggum fra Bekk Consulting. Intern veileder ved Institutt for informatikk ved Universitetet i Oslo var Dag Langmyhr.

Jeg ønsker å takke hovedveileder Thomas Johan Eggum, som har bidratt med stor faglig innsikt og gode råd gjennom hele prosessen. En stor takk også til Dag Langmyhr, som har bidratt med verdifulle innspill og meget solide LaTeX-kunnskaper.

1.1.1 Problemstilling

Selskapet Macromedia tok i 2002 i bruk uttrykket «Rich Internet Applications» for første gang, og definerte slike applikasjoner som «applikasjoner som innehar funksjonalitet tidligere forbeholdt tradisjonelle desktop-applikasjoner» [2]. Macromedia beskrev i samme artikkel hvordan man kunne lage rike internettapplikasjoner ved hjelp av deres Flash-teknologi. Rike internettapplikasjoner er med andre ord relativt ny teknologi. Det har siden den gang blitt definert flere teknikker og kommet til en rekke rammeverk som gjør det mulig å utvikle webapplikasjoner med funksjonalitet som gjør at de kan kalles RIAer.

Problemstillingen i denne masteroppgaven dreier seg om sikkerhetsaspektet ved denne nye typen teknologi og applikasjoner. Hvilke fallgruver må man som utvikler av slike applikasjoner passe seg for dersom man ønsker å unngå at feil i koden fører til sikkerhetshull? Finnes svakhe-

ter man har oppdaget i tradisjonelle webapplikasjoner i disse nye teknologiene og rammeverkene? Er det eventuelt introdusert noen nye?

For å oppnå innsikt i teknologien og faglig grunnlag for å kunne besvare disse spørsmålene ble det som en del av denne oppgaven utviklet to rike internettapplikasjoner og én serversideapplikasjon til bruk ved sikkerhetstesting. Det ble deretter studert hvilke sikkerhetsutfordringer som er mest vanlig i tradisjonelle webapplikasjoner, og undersøkt om disse er relevante også for RIAer. Det ble også undersøkt nærmere om bruk av RIA-rammeverk eventuelt innfører noen nye sikkerhetsutfordringer.

Hensikten med å undersøke disse problemstillingene er å komme fram til gode råd og huskeregler for RIA-utviklere. Slike såkalte «best practices» vil kunne gjøre det enklere for utviklere å lage sikre RIAer. Huskereglene vil bli oppsummert i en egen huskeliste for sikker utvikling av RIAer.

1.2 Definisjoner og forkortelser

1.2.1 Definisjoner

Uttrykk	**Definisjon**
Cookie	Informasjonskapsel lagret på brukerens pc.
Skrivebordsapplikasjon	Enkeltstående installasjon av programvare på en brukers pc.
Nettleserutvidelse / plugin	Programvare, ofte 3. parts, som utvider nettleserens funksjonalitet.
Obfuskert kode	Kildekode som er forsøkt gjort uleselig eller vanskelig å forstå.
Postback	Forespørsel fra klient til server hvor serveren etter prosessering sender hele websiden tilbake til klienten på ny.
Sesjons-id	Data brukt til å identifisere en sesjon.
Webservice	Programvare designet for å støtte maskin-til-maskin-interaksjon via nettverk.

Tabell 1.1: Definisjoner

1.2.2 Forkortelser

Forkortelse	Forklaring
CA	Certification Authority
DNS	Domain Name System
GWT	Google Web Toolkit
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JRE	Java Runtime Environment
JVM	Java Virtual Machine
MITM	Man-in-the-middle (angrepstype)
REST	Representational state transfer
RIA	Rich Internet Application
SDK	Software Development Kit
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
TLS	Transport Layer Security
URL	Uniform Resource Locator
XHR	XML Http Request
XML	Extensible Markup Language
XSRF	Cross-Site Request Forgery
XSS	Cross-Site Scripting

Tabell 1.2: Forkortelser

Kapittel 2

Bakgrunnsstoff

Det følgende kapittelet vil ta for seg bakgrunnsstoff for temaene webapplikasjoner og rike internettapplikasjoner. Det vil først bli gitt en nærmere forklaring på hva begrepet «webapplikasjon» vil si i kontekst av denne oppgaven. Deretter vil vi, for å få en bedre oversikt over det tekniske rundt hvordan webteknologi fungerer, se nærmere på HyperText Transfer Protocol (HTTP). Videre vil det bli gitt en forklaring på hva rike internettapplikasjoner er og vist eksempler på slike applikasjoner. Til slutt vil det bli presentert et utvalg aktuelle rammeverk som gjør det mulig å utvikle rike internettapplikasjoner.

2.1 Webapplikasjoner

En webapplikasjon lar brukerne utføre oppgaver og nyttiggjøre seg applikasjonen over internett, som oftest ved hjelp av en nettleser. Ofte har applikasjonene levet et tidligere liv som tradisjonelle, enkeltstående installasjoner på brukernes datamaskiner, for så å bli kodet om til webapplikasjoner. Fordelene med webapplikasjoner kontra tradisjonelle applikasjoner er mange. For eksempel har man tilgang til en webapplikasjon fra hvor som helst i verden så lenge man har internett-tilgang. I tillegg kjører applikasjonen sentralt på én maskin, noe som vil være en fordel ved oppdatering av applikasjonen og andre driftsrelaterte aspekter. Man slipper for eksempel unna med å oppdatere én server fremfor å måtte oppdatere et titalls eller kanskje til og med hundrevis av enkeltstående installasjoner hos hver av brukerne.

I tillegg til de nevnte fordelene vil en webapplikasjon også ha et fortrinn når det gjelder samarbeid mellom brukerne av applikasjonene og lagring av data. Istedenfor at applikasjonen er knyttet til en enkelt lokal

database, vil webapplikasjonen typisk vedlikeholde en sentral database. Dette forenkler driftsoppgaver som oppdatering og sikkerhetskopiering, i tillegg til at det åpner for at brukerne kan ha tilgang til de samme dataene samtidig - og at disse dataene til enhver tid vil være oppdatert.

Webapplikasjoner kan være så mangt, for å nevne noen eksempler kan slike applikasjoner være e-postklienter, booking- og reservasjonssystemer, dokument- og prosjekthåndteringssystemer, økonomisystemer og lignende.

I denne oppgaven defineres en webapplikasjon til å være en applikasjon brukere kan nyttiggjøre seg av over internett ved hjelp av en nettleser. Webapplikasjonen har et brukergrensesnitt som er synlig i brukernes nettleser. Beregninger og lagring kan skje både på klient- og serversiden.

2.1.1 HyperText Transfer Protocol (HTTP)

Internett benytter hovedsaklig protokollen HyperText Transfer Protocol (HTTP) i applikasjonslaget. Protokollen spesifiserer hvordan klienter og tjenere som benytter HTTP skal utveklse informasjon [14]. Protokollen beskriver hvilken struktur meldingene som utgjør kommunikasjonen skal ha, samt hvordan de skal overføres. Nettlesere som for eksempel Mozilla Firefox og Microsoft Internet Explorer utgjør klientsiden og webservere som for eksempel Apache eller Microsoft Internet Information Services utgjør tjenersiden. HTTP beskriver hvordan forespørselen en nettleser sender skal se ut og hvordan webservere svarer på disse ved å overføre nettsider tilbake til klienten. Ved bruk av HTTP er det klienten som tar kontakt med tjeneren/webserveren og ber om å få tilsendt informasjon [11].

En viktig egenskap ved HTTP er at det verken lagres eller tas vare på informasjon om klientene og deres forespørsler. Derfor kalles protokollen tilstandsløs.

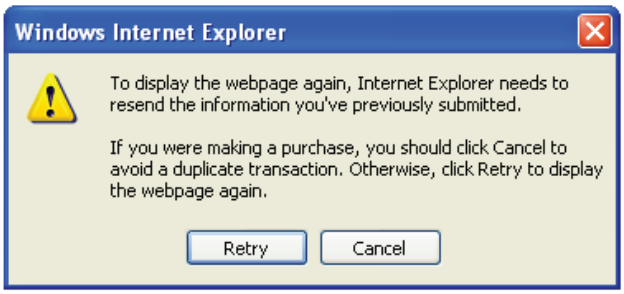
De to mest vanlige formene for forespørsler (engelsk: requests) fra klienter til tjenere er GET og POST. GET brukes når klienten forespør informasjon. Eventuelle parametre sendes da med som en del av en URL og er dermed synlige i nettleserens adressefelt. Dersom brukeren av en søketjeneste søker etter dokumenter som inneholder uttrykket «java», kan URL se for eksempel slik ut:

```
http://www.søkemotor.no/søk?java
```

Som vi ser er parameteren «søk» sendt med som en del av URL, og inneholder teksten «java». Et eksempel på en fullstendig GET-request kan se slik ut:

```
GET / HTTP/1.0
Host: www.ifi.uio.no
User-Agent: Mozilla/5.0
            (Windows; U; Windows NT 5.1; nb-NO; rv:1.9.0.8)
            Gecko/2009032609 Firefox/3.0.8
Accept: text/html,
        application/xhtml+xml,
        application/xml;q=0.9,*/*;q=0.8
Accept-Language: nb,no;q=0.8,
                 nn;q=0.6,en-us;q=0.4,
                 en;q=0.2
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
If-Modified-Since: Wed, 10 Dec 2008 08:50:49 GMT
If-None-Match: "c029-135d-45dad5b47cc40"
Cache-Control: max-age=0
```

POST bør benyttes når forespørselen eller handlingen brukeren ønsker å utføre medfører endring på serveren. Dette kan for eksempel oppstå når en bruker går fra tilstanden uregistrert til registrert eller fra utlogget til innlogget. I motsetning til hva som er tilfelle ved bruk av GET sendes ikke parametre med som en del av URL når man bruker POST. Eventuelle parametre sendes istedenfor med som en del av HTTP-forespørselen til serveren [11].



Figur 2.1: Internet Explorer: POST-forespørsel

Bruk av POST fører i de fleste nettlesere også til at brukeren blir bedt om bekreftelse dersom han/hun forsøker å sende den samme forespørselen flere ganger (se figur 2.1). Her følger et eksempel på hvordan en POST-forespørsel kan se ut:

```
POST / login.php HTTP/1.0
Host: webmail.uio.no
User-Agent: Mozilla/5.0
            (Windows; U; Windows NT 5.1; nb-NO; rv:1.9.0.8)
            Gecko/2009032609 Firefox/3.0.8
Accept: text/html,application/xhtml+xml,
        application/xml;q=0.9,*/*;q=0.8
```

```
Accept-Language: nb,no;q=0.8,nn;q=0.6,
                 en-us;q=0.4,en;q=0.2
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://webmail.uio.no/src/login.php
Cookie: SQMSESSID=8t354db3b1njmcfiio8ais49042sgupj;
        squirrelmail_language=deleted;
        SQMSESSID=8t354db3b1njmcf iio8ais49042sgupj
```

Referer-header

Majoriteten av nettlesere sender i de fleste HTTP-forespørsler med en linje som kalles «Referer» [11]. Når brukeren klikker på en lenke i for eksempel et HTML-dokument, inkluderes en referanse til dette dokumentets URL i forespørselen som sendes for å etterspørre det nye dokumentet. Denne linja i forespørselen kan for eksempel se slik ut:

```
Referer: http://www.eksempel.com/index.html
```

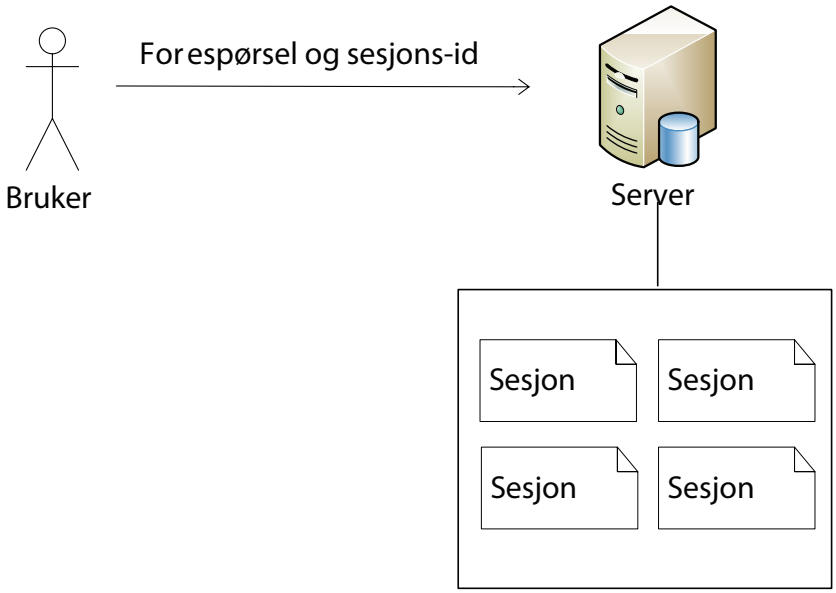
Referer-headeren egner seg allikevel ikke som en sikkerhetsmekanisme for å kontrollere hvor forespørsler har sitt opphav. Årsaken til dette er at forespørslene som referer-headeren er en del av, kommer fra klientsiden. Det er mulig å endre headeren på klientsiden. Dette gjør at man som utvikler må ta høyde for at referer-headeren kan være misvisende.

Hvordan omgå tilstandsløshet i HTTP

Mange webapplikasjoner har et behov for å ivareta tilstand knyttet til hver enkelt bruker. I utgangspunktet er HTTP som nevnt i 2.1.1 tilstandsløs, og tilbyr ingen slik funksjonalitet. Med tilstand menes informasjon om brukeren, som for eksempel hvorvidt en bruker er innlogget eller ei. En måte å omgå dette problemet på er å benytte såkalte «cookies». Prinsippet bak en slik løsning er at tilstand implementeres ved at tjeneren ber klienten ta vare på en gitt verdi (se figur 2.2). Denne verdien kalles ofte en sesjons-id («session-id» på engelsk). Klienten sender med den aktuelle verdien i hver forespørsel til tjeneren, og gjør det på denne måten mulig å hente fram lagret informasjon om den aktuelle brukeren og dermed ivareta tilstand.

2.1.2 HTTPS

HTTPS er HTTP-kommunikasjon over en kryptert kommunikasjonskanal [11]. Den krypterte kommunikasjonen skjer over protokollen Secure



Figur 2.2: Hvordan omgå tilstandsløshet med sesjoner

Socket Layer (SSL), som opprinnelig ble designet av Netscape [14]. Alternativt sendes kommunikasjonen over SSLs etterfølger Transport Layer Security (TLS). TLS er en modifisert versjon av SSL versjon 3 [14].

Kommunikasjon over SSL/TLS kan deles inn i tre forskjellige faser [14]:

- Handshake
- Enighet om krypteringsnøkkel
- Dataoverføring

Det kan også legges til et fjerde steg hvor klienten validerer serveren, men dette steget er ikke påkrevet.

Hvis en bruker kontakter for eksempel en nettbank over HTTPS, vil første steg være at klient og server blir enige om hvilken krypterings-algoritme som skal brukes. Deretter sannsynliggjør nettbanken overfor brukeren at nettbanken virkelig er den nettbanken den utgir seg for å være. Dette skjer ved at serveren sender klienten et sertifikat. Sertifikater utstedes av såkalte «Certification authorities» (CA), sertifiseringsmyndig-heter. Moderne nettlesere kommer med en ferdig definert liste over CAs nettleseren stoler på, og det er denne lista klienten tar utgangspunkt i når sertifikatet fra serveren skal valideres.

Til tross for kryptert dataoverføring vil allikevel følgende sikkerhetsutfordringer være til stede ved bruk av HTTPS [11]:

- Dersom sertifikatet nettleseren mottar fra serveren er usignert, utløpt eller av andre grunner ikke aksepteres, vil nettleseren spørre brukeren om hva som skal skje videre. Dette åpner for at brukeren tar feil avgjørelse og godtar et usikkert sertifikat.
- Sertifiseringsmyndigheter kan bli lurt til å utstede falske sertifikater.
- Nettleserens liste over trygge sertifikater kan være uriktig eller utdatert.
- Implementasjonen av SSL/TLS i nettleseren eller på serveren kan inneholde feil eller mangler.

Kommunikasjon over SSL/TLS er ansett for å være sikker [11]. Det vil være vanskelig for en angriper å angripe selve kommunikasjonskanalen ettersom den er kryptert, men dataoverføringen er også det eneste som sikres. Det vil fortsatt være mulig for en angriper å angripe både server og klient [11]. Angripere velger som oftest det svakeste leddet, som i dette tilfellet ofte vil være brukeren.

2.2 Rike internettapplikasjoner

Uttrykket «Rike internettapplikasjoner» er et diffust uttrykk. Det er vanskelig å fastslå hva som er kriteriene for at en webapplikasjon skal være en RIA, og det er uklart hvilke teknologier man må benytte for at en applikasjon skal fortjene å falle inn under en slik definisjon.

Rike internettapplikasjoner er, som navnet antyder, en form for webapplikasjoner. Den enkle forklaringen på hva som er forskjellen mellom en rik internettapplikasjon og en tradisjonell, er at en rik internettapplikasjon innehar funksjonalitet og utseende som tildligere har vært forbeholdt tradisjonelle skrivebordsapplikasjoner [2]. RIAer har ofte et rikere innhold i form av media (lyd, bilde, video).

Den noe mer detaljerte og tekniske forklaringen er at der en tradisjonell internettapplikasjon lager alt innholdet på serversiden og sender tilbake til nettleseren i form av HTML (HyperText Markup Language), åpner RIAer for at det kan skje beregninger og midlertidig lagring på klientsiden. Fordelene med sistnevnte måte å gjøre det på er flere og svært nyttige i applikasjoner hvor man ønsker en rikere brukeropplevelse enn hva man

hadde i tradisjonelle, HTML-baserte internettapplikasjoner. Ved bruk av statisk HTML måtte hele siden genereres på nytt på tjeneren for hver eneste tilbakemelding eller postback fra brukeren. Dette fører til lang responstid, og fører til at internettapplikasjonen framstår som treg og tungvinn for brukeren. Ved å kunne utføre beregninger og lagre noe data midlertidig hos klienten, kan man lage applikasjoner som fremstår som raske sammenlignet med tradisjonelle webapplikasjoner. Brukeren slipper i større grad unna ventetiden som oppstår mellom en postback sendes og nettleseren mottar den nye siden.

Rike internettapplikasjoner benytter en rekke forskjellige teknikker for å skape bedre brukeropplevelse i form av rikere innhold og hurtigere respons på brukernes handlinger. Hvilke teknikker rammeverkene benytter varierer fra teknologi til teknologi, og det er vanskelig å gi en eksakt definisjon av hva et rammeverk må tilby for at det skal være egnet til å utvikle RIAer i. Fellesnevneren for RIAer er uansett at de foretar en form for beregning eller lagring på klientsiden. For å få til dette må det kjøres kode på klientsiden.

2.2.1 Eksempler på rike internettapplikasjoner

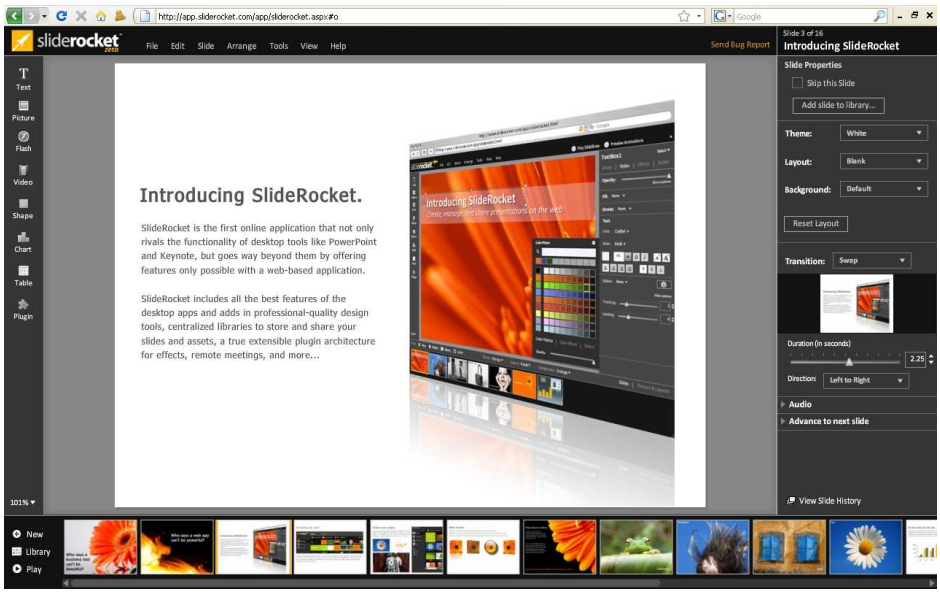
I det følgende vil det bli gitt tre eksempler på rike internettapplikasjoner. Hensikten er å gi leseren bedre forståelse av hva som kjennetegner en slik applikasjon, og hva som skiller den fra en tradisjonell webapplikasjon.

Sliderocket

Sliderocket er en rik internettapplikasjon som brukes til å lage presentasjoner. Applikasjonen kjører i nettleseren, og benytter nettleserutvidelsen Adobe Flash Player. Sliderocket er laget ved hjelp av rammeverket Adobe Flex, som presenteres på side 14. Figur 2.3 viser hvordan det ser ut når man redigerer en side i en presentasjon man har laget ved hjelp av Sliderocket.

Denne applikasjonen er et spesielt godt eksempel på en RIA fordi den tilbyr samme funksjonalitet som en tradisjonell skrivebordsapplikasjon, i dette tilfellet Microsoft Office PowerPoint. Sliderocket tilbyr samme funksjonalitet på følgende områder:

- Det er mulig å opprette, editere og lagre presentasjoner bestående av flere sider, eller såkalte slides.
- Applikasjonen har funksjonalitet for å legge på transisjoner ved sidebytte i presentasjonen.



Figur 2.3: Sliderocket

- Applikasjonen har mulighet for å legge til bilder, lyd og video i presentasjoner.
- Det er mulig å vise presentasjoner i fullskjerm ved hjelp av applikasjonen.

I tillegg har SlideRocket funksjonalitet som gjør det mulig å importere PowerPoint-presentasjoner man allerede har laget. Alt dette foregår i brukerens nettleser, og brukeren installerer ingen form for programvare utover Adobe Flash Player for å få tilgang til applikasjonen. Presentasjonene lagres på nett, og brukeren har dermed tilgang til å hente fram, vise og editere presentasjoner fra hvor som helst i verden hvor det finnes internettaksess. Det eksisterer en tilsvarende RIA for tekstbehandling, Buzzword. Buzzword er laget av Adobe og i likhet med Sliderocket utviklet ved hjelp av rammeverket Flex.

Scrapblog

På www.scrapblog.com kan brukerne opprette såkalte scrapbooks. Dette er bøker eller fotoalbum som brukerne selv designer. Scrapblog er i likhet med Sliderocket en typisk RIA fordi den tillater brukerne å legge til forskjellige typer media (bilder, lyd og video) og innehar funksjonalitet

Figur 2.4: Scrapblog

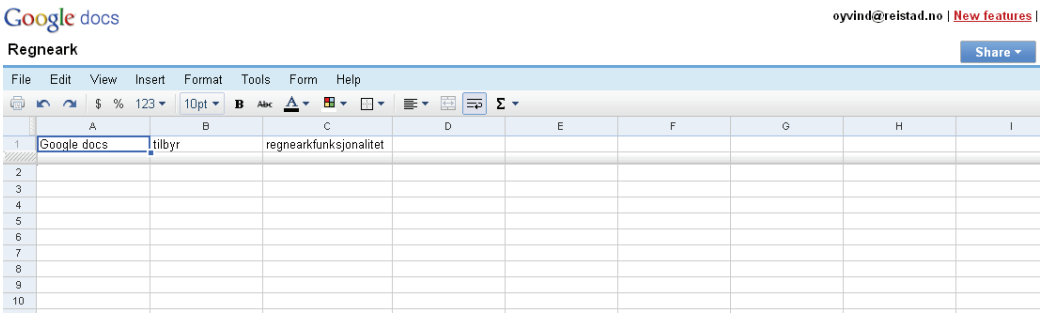
man tidligere har måttet benytte skrivebordsapplikasjoner for å få tilgang til.

Scrapblog er også laget ved hjelp av Adobe Flex-rammeverket, vises i nettleseren og krever Adobe Flash Player-utvidelse.

Google docs

Google docs er en RIA utviklet av Google. Bruk av applikasjonen er gratis og den er tilgjengelig for alle som har en Google-konto. Applikasjonen inneholder tekstbehandler, regneark og verktøy for å lage presentasjoner. Man kan lagre dokumenter på Googles server, og om ønskelig dele dokumenter med andre brukere. En autolagringsfunksjon sørger for at brukerne mister så lite data som mulig dersom for eksempel internettforbindelsen skulle bli brutt hos klienten. Applikasjonen benytter blant annet JavaScript for å gi brukerne opplevelsen av at applikasjonen fungerer på samme måte som en tradisjonell skrivebordsapplikasjon.

Google docs oppfyller kravene til en rik internettapplikasjon ikke bare fordi den tilbyr funksjonalitet fra skrivebordsapplikasjoner, men også fordi den tilbyr en rekke andre fordeler som mulighet for å dele dokumenter med andre brukere, lagring på server og autolagringsfunksjon.



Figur 2.5: Google Docs Spreadsheet

2.2.2 Rammeverk, teknologier og teknikker

Det finnes flere ulike rammeverk, teknologier og teknikker som gjør det mulig å konstruere rike internettapplikasjoner. Jeg vil i det følgende kort presentere et utvalg av disse.

Adobe Flex

Flex er et open-source rammeverk for å lage RIAer, og et produkt fra programvareleverandøren Adobe. Flex-applikasjoner vises i brukernes nettlesere ved hjelp av nettleserutvidelsen Flash Player. Denne nettleserutvidelsen er en forutsetning for å kunne benytte Flex-applikasjoner. Flex-rammeverket bygger på Flash-teknologien, som stammer fra samme programvareleverandør. Det er umulig å slå fast eksakt hvor mange av nettets brukere som har denne utvidelsen installert i sin nettleser, men i en markedsundersøkelse foretatt av Millward Brown/Lightspeed Research hevdes det at ca. 99 % av internettbrukere har utvidelsen installert [1] hvis man regner med alle versjoner. Den nyeste versjonen, Flash Player 10, var i følge undersøkelsen i mars 2009 installert på cirka 75% av nettleserne i Europa. Disse tallene antyder at man vil kunne være ganske sikker på at de fleste brukere vil kunne benytte seg av en slik type applikasjon selv om applikasjonene krever Flash Player-utvidelsen.

Flex-applikasjoner utvikles ved hjelp av det deklarative, XML-baserte språket MXML. Det innebærer at man kan spesifisere applikasjonens grafiske brukergrensesnitt ved hjelp av tags på samme måte som i HTML. For å skrive programkode for klientsiden benyttes det objektorienterte, ECMAScript-baserte scriptingspråket ActionScript 3 [7].

Flex analyserer MXML-tagger og kompilerer en SWF-fil (uttales «swiff») som inneholder de korresponderende ActionScript-objektene [7].

Figur 2.6: Adobe Flex-logo

Swf formatet benyttes også av applikasjoner eller animasjoner laget med Adobes Flash-teknologi.

Rammeverket inneholder en rekke ferdige komponenter som er ment å skulle gjøre det raskere å utvikle avanserte applikasjoner. Utviklere kan konstruere grafiske brukergrensesnitt og inkludere forskjellige brukergrensesnittkomponenter etter dra-og-slipp metoden. Eksempler på nyttige komponenter inkludert i Flex er riktekstеditor, progressbar og videoavspillerkomponent. Utvikling av Flex-applikasjoner kan gjøres i Adobes eget verktøy Flexbuilder 3, som er basert på det kjente programmeringsmiljøet eclipse, men det er også mulig å benytte andre editorer.

Ajax (Asynkront JavaScript og XML)

Ajax er ikke et rammeverk eller en teknologi, men en teknikk for å sende asynkrone kall til serversiden ved hjelp av JavaScript og XML. Ajax gjør det mulig for utviklere å lage rike internettapplikasjoner fordi asynkrone kall og JavaScript gjør det mulig å dynamisk endre nettsider som vises i brukerens nettleser.

Applikasjoner som benytter AJAX krever, ettersom de er basert på JavaScript og tradisjonell HTML, ingen nettleserutvidelse. Den tekniske forklaringen bak de asynkrone kallene teknikken benytter finner vi i XMLHttpRequest-objektet (forkortes XHR), som gjør det mulig å programmatisk sende GET og POST-forespørsler i JavaScript-kode. Dette gjør det mulig å lage mer interaktive og responsive webapplikasjoner.

Google Web Toolkit

Google Web Toolkit (forkortes GWT) er et rammeverk fra Google som forenkler bruk av AJAX. Rammeverket gjør det mulig for utviklere å skrive klientsidekode i programmeringsspråket Java, kode som deretter kompileres til optimalisert JavaScript. Nettlesere tolker JavaScript ulikt,

og et av Googles argumenter for å benytte GWT er at man unngår å måtte skrive samme AJAX-applikasjon flere ganger for å kunne støtte ulike nettlesere. Google hevder at rammeverket skal sørge for at den ferdig kompilerte JavaScript-koden fungerer på tvers av de fleste nettlesere. Det er også mulig å debugge Java-koden man skriver i GWT-rammeverket som om den var ordinær Java-kode. En GWT-applikasjon trenger, av samme årsaker som AJAX, ikke en egen nettleserutvidelse for å kunne vises. GWT har allikevel enkelte begrensninger, ettersom hva som er

Figur 2.7: Google Web Toolkit

programmatisk mulig i GWT begrenses av hva som er mulig å få til i JavaScript. Rammeverket støtter mye av hva som er mulig å gjøre i Java, men ikke alt lar seg kompilere til Javascript og er følgelig ikke mulig å få til.

Java Applet

En Java Applet er en frittstående Java-applikasjon som kjører i nettleseren. Denne typen applikasjoner ble introdusert i den første versjonen av Java-språket i 1995, og gjør det mulig å kjøre Java-kode på klientsiden. Det skjer ved at en applet kjører i en nettleser ved hjelp av Java Virtual Machine (JVM). Java Applet var således en forgjenger for rike internettapplikasjoner. Siden Javas bytekode kjøres i JVM og er plattformuavhengig, kan Java Applets kjøre i nettlesere på forskjellige operativsystemer, for eksempel Windows, Linux og Mac OS.

Java Applets kjører i de fleste nettlesere i en såkalt sandkasse (engelsk: sandbox) som skal forhindre tilgang til ressurser som for eksempel filsystemet på klientsiden. Det er imidlertid mulig å gi en Applet utvidet tilgang på klientsiden dersom brukeren samtykker til det.

En av ulempene med tidlige versjoner av Java Applets var at hver nettleserleverandør implementerte sin egen versjon av JVM. For å sikre kompatibilitet måtte derfor utviklere av Java Applets skrive programmene slik at de kunne kjøre på alle de ulike JVM-implementasjonene [24].

På grunn av den unge Java-spesifikasjonen og de mange forskjellige JVM-implementasjonene ble Java Applets velkjente for sine omfattende sikkerhetproblemer [24]. Senere har man tatt i bruk Java Plugin, hvor man går bort fra nettleserspesifikk implementasjon av JVM. Ved bruk av Java Plugin er det nå også mulig for utvikleren å angi hvilken versjon JVM som kreves for at applikasjonen skal kunne kjøre.

JavaFX

Det amerikanske selskapet Sun lanserte i desember 2008 en egen plattform for utvikling av RIAer, JavaFX (Sun ble forøvrig kjøpt opp av Oracle 20. april 2009). Plattformen benytter i likhet med Silverlight og Flex et deklarativt scriptspråk, i JavaFX kalles dette JavaFX Script. For å kunne kjøre JavaFX-applikasjoner i nettleseren må brukeren ha nettleserutvidelsen Java Plugin installert.

Figur 2.8: JavaFX-logo

JavaFX er integrert med JRE (Java Runtime Environment), og det er mulig å «dra» en JavaFX-applikasjon fra nettleseren til skrivebordsbakgrunnen for å installere applikasjonen som en frittstående applikasjon (dette forutsetter at brukeren har Java SE 6 update 10 eller nyere). Det er også mulig å kjøre JavaFX-applikasjoner på mobiltelefoner med Java Mobile Emulator-støtte.

Nåværende versjon, JavaFX 1.2, kom i juni 2009. Det er varslet at senere versjoner skal kunne kjøres på et enda større utvalg av enheter med Java-støtte, for eksempel Blu-Ray spillere.

Microsoft Silverlight

Silverlight er et rammeverk fra den kjente programvareleverandøren Microsoft og benyttes til utvikling av RIAer. Silverlight benytter i likhet med Flex en utvidelse til brukerens nettleser, og gjør det mulig å vise vektorgrafikk, animasjoner, lyd, bilde og video. Versjon 1 ble tilgjengelig i april 2007. Versjon 2 kom i oktober 2008, og har forbedret multimediastøtte samt inkluderer deler av C#-bibliotekene som finnes i Microsofts .NET-

rammeverk. Microsoft offentliggjorde versjon 3 av Silverlight 9. juli 2009.

Figur 2.9: Microsoft Silverlight-logo

Nøyaktig hvor stor utbredelse Silverlights nettleserutvidelse har er vanskelig å finne et entydig og upartisk svar på. Microsoft hevder at utbredelsen i USA er opp mot 30%[15], nettsiden RIAstats.com hevder at Silverlight versjon 3 har 19% utbredelse og at versjon 2 har 6% utbredelse [10]. Selv om anslagene er svært ulike, gir de en klar indikasjon på at Silverlight-utvidelsens utbredelse er langt lavere enn utbredelsen Flash Player har.

Brukergrensesnittet i Silverlight-applikasjoner defineres i det XML-baserte filformatet XAML. Når det gjelder bakenforliggende kode kan utviklere selv velge hvilket programmeringsspråk fra .NET-plattformen de ønsker å benytte, for eksempel C#, VB, Ruby eller Python. Silverlight-applikasjoner utvikles i Microsofts eget utviklingsverktøy Visual Studio eller andre utviklingsverktøy som støtter .NET-plattformen.

2.2.3 Oppsummering

Rammeverk/teknikk	Krever nettleserutvidelse (plugin)	Open source
Adobe Flex	Flash Player	Ja[1]
Ajax	Nei	-
Google Web Toolkit	Nei	Ja
Java Applet	Java plugin	Ja
JavaFX	Java plugin	Ja
Microsoft Silverlight	Silverlight plugin	Nei

Tabell 2.1: RIA-rammeverk, teknologier og teknikker

2.2.4 Plattformkompatibilitet

Når det gjelder plattformkompatibilitet støtter både Adobe Flex/Flash Player, AJAX, JavaFX, Java Applet og Google Web Toolkit alle sammen Microsoft Windows, Linux og Mac-plattformen. Microsoft Silverlight støtter bare delvis Mac, og støtter dessuten Linux kun gjennom et open-source prosjekt kalt Moonlight. Moonlight har foreløpig ikke full støtte for Silverlight versjon 2.

Rammeverk/teknikk	Windows	Linux	Mac
Adobe Flex	Ok	Ok	Ok
Ajax	Ok	Ok	Ok
GWT	Ok	Ok	Ok
Java Applet	Ok	Ok	Ok
JavaFX	Ok	Ok	Ok
Microsoft Silverlight	Ok	Moonlight[2]	Ok[3]

Tabell 2.2: RIA-rammeverkenes plattformkompatibilitet

[1] Adobe Flex SDK er gratis. Utviklingsmiljøet Adobe Flex Builder må kjøpes, men er ikke påkrevet.

[2] Moonlight er et open source-prosjekt som har som mål å utvikle en Silverlight-plugin for Linux.

[3] Versjon 2 og 3 er kun tilgjengelig for Mac med Intel-prosessor.

Kapittel 3

Sikkerhet i webapplikasjoner

I dette kapittelet vil vi se nærmere på sikkerhet i informasjonssystemer generelt og sikkerhet i webapplikasjoner og rike internettapplikasjoner spesielt. Kapittelet vil omhandle viktigheten av og motivasjon for sikkerhet ved å se nærmere på historien og hvilket trusselbilde man som utvikler står overfor.

3.1 Hvorfor er sikkerhet viktig?

For å motivere for og forklare at sikkerhet er viktig er det naturlig å forsøke å gi svar på følgende spørsmål:

- Hva er sikkerhet?
- Hvorfor er sikkerhet viktig?
- Hvilken relevans har sikkerhet for rike internettapplikasjoner?

Det er vanskelig å finne ett enkelt fasitsvar på disse spørsmålene. Sikkerhet kan være mange ting, og det er flere årsaker til hvorfor dette er et viktig aspekt ved en applikasjon. En av de mest kjente og aksepterte modellene for informasjonssikkerhet kalles «the CIA triad» [23], som nevner de tre aspektene Confidentiality, Integrity og Availability. For å illustrere hvorfor disse faktorene er sentrale vil hver av dem bli relatert til nytten de har i en velkjent applikasjon, for eksempel en nettbank. Årsaken til at disse aspektene er egnet til å motivere for sikkerhet er:

- For det første nevnes tilgjengelighet (availability) som et mål for et IT-system. Med dette menes at informasjonen i IT-systemene bør

være mest mulig tilgjengelig i den forstand det ikke bør oppstå situasjoner som gjør informasjonen utilgjengelig for brukerne. Eksempler på slike situasjoner er feil eller mangler i systemet som kan forårsake ødeleggelse av data eller opphold i tilgjengeligheten. Utilgjengelige data eller en utilgjengelig applikasjon vil hindre brukerne i å utføre sine gjøremål i applikasjonen. Sett i perspektiv av eksempelet nettbank er tilgjengelighet en sentral faktor for at en slik applikasjon skal kunne fungere og bli brukt.

- For det andre nevnes integritet (integrity). Integritet er et ord som stammer fra det latinske uttrykket «integritas», som i sin opprinnelige betydning sto for «sunnhet» eller «uskadd tilstand» [6]. I denne sammenhengen betyr det at dataene i applikasjonen må ha integritet, altså at det ikke skal være mulig å endre dataene dersom man er en utenforstående som ikke er ment å skulle ha tilgang. Et system som ikke innehar integritet vil inneholde data man ikke kan stole på. I en nettbank er integritet et ufravikelig krav, da både banken og kundene er avhengige av at data i en slik applikasjon (for eksempel låne- og kontosaldo) ikke kan endres av angripere, men til enhver tid er korrekt.

- For det tredje tar definisjonen opp konfidensialitet (confidentiality). Hvis en webapplikasjon behandler data man ikke ønsker at skal være tilgjengelige for alle og enhver, ønsker man at dataene skal behandles konfidensielt. En rekke IT-systemer inneholder slik sensitiv informasjon, derfor er dette punktet en svært viktig motivasjonsfaktor for IT-sikkerhet. I nettbank-eksempelet er konfidensialitet viktig på flere måter, for eksempel ved at utenforstående ikke skal ha anledning til å se konto- og lånesaldo, transaksjoner og så videre.

De tre aspektene i «the CIA triad» belyser viktige mål ved informasjonssikkerhet. Disse målene påvirker hverandre, og kan dermed komme i konflikt. For eksempel kan stort fokus på konfidensialitet komme i konflikt med god tilgjengelighet og vice versa. En nettbank med en enkel autentiseringsmekanisme som bare krever brukernavn og passord vil ha stor grad av tilgjengelighet, men en slik løsning vil gått på bekostning av konfidensialitet og integritet på grunn av at angripere lett vil kunne utnytte den høye tilgjengeligheten. Derfor må de tre aspektene i «the CIA triad» avpasses hverandre.

Sikkerhet i IT-sammenheng kan sees fra forskjellige perspektiver. På lavere nivåer er det viktig å ha god sikkerhet i infrastruktur, som går på

det å ha en god brannmur, oppdatert programvare (webservere, operativsystemer og lignende) og sikre kommunikasjonskanaler (kryptering). På høyere nivå har man sikkerhet i selve applikasjonene og programkoden. Det hjelper lite med en god brannmur hvis inntrengeren allikevel kan få tak i konfidensielle data ved for eksempel å sende applikasjonen uventede inputdata (se for eksempel SQL-injection på side 26).

Hvorfor er sikkerhet viktig? Som det fremgår av drøftingen over er tilgjengelighet, integritet og konfidensialitet svært viktige faktorer for informasjonssystemer. Årsakene til dette er flere. På den ene siden finnes det lover og regler som sier noe om hvordan informasjon i informasjonssystemer skal behandles, og at man ikke uten videre kan spre oppsamlet og potensielt sensitiv informasjon om brukerne uten samtykke. På den andre siden ønsker svært få brukere å benytte IT-systemer som inneholder uriktige data, hvor konfidensiell informasjon ligger åpent og som på toppen av det hele er utilgjengelige. Videre vil ingen kunder betale for slike systemer. God sikkerhet bør derfor være blant de mest sentrale egenskapene ved et informasjonssystem. I neste avsnitt vil det bli gitt eksempler på hendelser hvor sikkerheten i IT-systemene ikke har vært god nok.

3.1.1 Eksempler

Det finnes mange eksempler på mangelfull sikkerhet i webapplikasjoner. For å understreke viktigheten av sikkerhet presenteres det her et lite utvalg saker fra media:

- «Hacket partikkelakselleratoren på CERN» [17]. En gresk hackergruppe brøt seg inn i en webserver hos CERN (The European Organization for Nuclear Research) i september 2008. Angriperne plasserte en beskjed på en av CERNs nettsider. Den aktuelle webserveren ble benyttet til å vise resultater fra et eksperiment i forbindelse med partikkelakselleratoren Large Hadron Collider (LHC). Serveren var ikke direkte forbundet LHC og angrepet forårsaket ingen skade, men en talsmann for CERN sier at hendelsen har fått dem til å sette søkelys på sikkerhet ved CERN.

- «60 000 personnumre på avveie» [19]. Grunnet en feil i en webapplikasjon hos Tele2, var det en periode mulig å hente ut Tele2s register over personnumre. Registeret omfattet 60 000 personnumre. Datatilsynets direktør Georg Apenes var en av de som opplevde at personnummeret kom på avveie. I følge artikkelen oppdaget Apenes

identitetstyveriet når han mottok et SIM-kort i posten i forbindelse med tegning av et nytt mobilabonnement. Apenes reagerte på dette, ettersom han ikke hadde bestilt noe slikt abonnement.

- «316 norske nettsteder hacket siste måned» [13]. Markus Harboe fra IT-sikkerhetsfirmaet mnemonic og Erlend Oftedal fra Bekk Consulting holdt under utviklerkonferansen JavaZone 2008 et foredrag om sikkerhet i webapplikasjoner. Harboe hevdet at 316 norske nettsteder hadde blitt hacket siste måned, og at sikkerheten generelt er nedslående. «Web er en veldig synlig, tilgjengelig og effektiv angrepsflate. Det blir stadig mer vanlig å bruke web for å angripe webbens brukere», ytret Harboe i foredraget.

3.2 Trusselbilde

Trusselbildet for webapplikasjoner og RIAer er sammensatt. Et nettsted er alltid tilgjengelig og dermed en utsatt angrepsflate. Angrep mot webapplikasjoner kan ha svært forskjellige opphav; alt fra såkalte «script kiddies», som er nybegynnere som baserer seg på andres arbeid, til større kriminelle organisasjoner som er ute etter profitt [21].

Det kan være flere årsaker til at noen ønsker å utfordre sikkerheten i webapplikasjoner. Noen gjør dette for moro skyld, andre er politisk motiverte og noen gjør det av økonomiske årsaker [9]. Jo høyere verdi en ressurs har for en angriper, jo mer sannsynlig er det at angriperen vil forsøke å angripe nettopp denne ressursen [9]. Tid til rådighet samt utstyrs- og kunnskapsnivå hos angriperen er også faktorer som spiller inn på hvor utsatt en ressurs er.

3.3 Tradisjonelle angrepsmetoder

Det finnes en rekke kjente angrepsmetoder for tradisjonelle webapplikasjoner. Det vil i dette delkapittelet bli gitt en beskrivelse av hvordan disse sikkerhetssvakhetene oppstår, og for hver av dem nevnes også helt kort hva man kan gjøre for å unngå dem.

Utvalget av tradisjonelle angrepsmetoder som presenteres her tar utgangspunkt i OWASP Top Ten [22]. Denne lista består av de ti mest vanlige sikkerhetssvakhetene i webapplikasjoner i 2007 og ble til ved at man hentet ut de ti mest vanlige websikkerhetssvakhetene fra MITRE Vulnerability Trends for 2006. OWASP står for The Open Web Application

Security Project og er en verdensomspennende non-profit-organisasjon som har som mål å bedre sikkerheten i webapplikasjoner ved å synliggjøre denne problematikken og bevisstgjøre utviklere.

Følgende sikkerhetssvakheter inngår i OWASPs liste:

Cross-site scripting (XSS)

Cross-site scripting går ut på å vise skadelig eller uønsket HTML i en webapplikasjon. Som oftest opptrer XSS-forsøk i form av et script som kjøres i nettleseren til brukeren av en webapplikasjon som har denne sikkerhetssvakheten [11]. Sårbarheten oppstår når det er mulig for angripere å sende inn HTML eller et script til en webapplikasjon, som applikasjonen senere ufiltrert viser andre brukere.

Det er vanlig å dele inn i to typer XSS; reflektert og persistent. Ved bruk av førstnevnte form for XSS lager angriperen en URL som inneholder for eksempel et script. Et eksempel på en slik url kan være:

```
http://www.webside.no/sok?ord=<script>alert('Hei!')</script>
```

Ofte vil søkestrengen vises som en del av nettsiden man får opp etter å ha sendt en slik forespørsel. Søkestrengen hentes i eksempelet over fra parameteren «ord». Dersom denne ikke filtreres, vil scriptet kjøre og vise meldingen «Hei!» i offerets nettleser. Angriperen må derfor ved bruk av reflektert XSS lure offeret til å klikke på en link slik at forespørselen sendes.

Persistent XSS er basert på at angriperen sender inn et script som lagres i applikasjonen. Dersom inputdataene fra angriperen senere vises til andre brukere og applikasjonen ikke filtrerer angriperens input, vil scriptet og angrepet ramme alle påfølgende brukere av applikasjonen.

Et eksempel på en applikasjon hvor persistent XSS kan oppstå er en enkel gjestebok. En slik applikasjon tilbyr funksjonalitet hvor besøkende på den aktuelle nettsiden kan legge igjen en hilsen. Istedenfor å skrive «Hei og gratulerer med flott hjemmeside», kan en angriper velge å for eksempel sende inn følgende script:

```
<script>alert('Hallo alle sammen!');</script>
```

Dersom den aktuelle applikasjonen er sårbar for persistent XSS, vil scriptet kjøre neste gang gjesteboken vises. Det fører til at alle påfølgende brukeres nettlesere åpner et popup-vindu med teksten «Hallo alle sammen!».

Scriptet i dette eksempelet er harmløst, men applikasjoner som tillater XSS åpner mange muligheter for angripere. Det er mulig for angripere å

lage script som skaffer dem for eksempel offerets cookie eller sesjons-id. Dette kan benyttes av angriperen til å utgi seg for å være brukeren i en webapplikasjon som i utgangspunktet krever innlogging.

Slike angrep er mulige i applikasjoner som ikke kontrollerer eller filtrerer innhold som sendes tilbake til brukerne. Problemet unngås ved å filtrere brukergenerert innhold.

SQL-injection

En artikkel utgitt i Phrack Magazine fra 1998 [18] er antatt å være en av de første skriftlige beskrivelsene av denne angrepsmetoden. Artikkelen tar opp problematikken med SQL-injections eller SQL-injiseringer. Denne fallgruven kan man som utvikler gå i hvis man unnlater å håndtere kontrolltegn, eller hvis man unnlater å håndtere kontrolltegn korrekt. Angripere som benytter seg av SQL-injisering blir, hvis svakheten er tilstede, istand til å modifisere databasespørringene fra en webapplikasjon [11].

Figur 3.1: Innloggingsskjema

Figur 3.1 viser et innloggingsskjema. Når brukeren trykker på «Logg inn»-knappen, hentes verdiene fra e-post- og passordfeltet. Verdiene brukes i følgende Java-kode på serversiden:

```
ResultSet rs = sql.executeQuery("SELECT * FROM users WHERE email='" + userName +
                                "'AND password='" + password + "'");
```

En angriper som benytter SQL-injection og kjenner e-postadressen til en bruker av systemet vil her kunne logge inn uten passordet. Dette kan gjøres ved å fylle inn for eksempel «bruker@ifi.uio.no» i e-post feltet og følgende tegn i passordfeltet:

hei' OR 1='1

SQL-spørringen som kjøres mot databasen vil da se slik ut:

```
SELECT * FROM users WHERE email = 'bruker@ifi.uio.no' AND
 password = 'hei' OR 1='1';
```

Som vi ser av spørringen har angriperen ved å benytte SQL-injection endret spørringen som kjøres mot databasen. Istedenfor å hente bruker fra databasen basert på e-post og passord, henter den nye spørringen ut brukeren dersom passordet er «hei» eller 1 = 1. Ettersom siste del av uttrykket alltid vil være sann, hentes brukeren fra databasen og angriperen er logget inn i applikasjonen som «bruker@ifi.uio.no».

Som man ser av eksempelet er det bruk av kontrolltegn som input til webapplikasjonen som gjør dette mulig. Angriperen former en ny eller endret databasespørring ved å injisere kontrolltegn og tekst som endrer den opprinnelige SQL-spørringen. Hvis utvikleren ikke har sørget for å håndtere eventuelle kontrolltegn korrekt i situasjoner hvor input brukes som en del av en SQL-spørring, risikerer man at en angriper i praksis kan modifisere databasespørringene fra webapplikasjonen etter eget forgodtbefinnende. Dette er svært uheldig, da man i så tilfelle risikerer at for eksempel brukernes brukernavn og passord eller annen sensitiv informasjon i databasen blir tilgjengelig for angripere. Samtidig åpner feilen for at angripere kan endre og slette data.

Måten å unngå problemet på er å håndtere spesialtegn slik at det blir umulig å endre spørringene. Det er da viktig å gjøre seg kjent med alle mulige kontrolltegn som støttes av databasehåndteringssystemet. Kontrolltegnene må håndteres/«escapes» slik at de mister sin funksjon som kontrolltegn. Et annet alternativ er å benytte såkalte «prepared statements» i SQL [11]. Prepared statements er databasespørringer som sendes til databaseserveren med inputparameterne adskilt fra selve spørringen, slik at spørringen i seg selv ikke kan endres.

Usikker direkte objektreferanse

En fallgruve utviklere iblant går i er bruk av usikre direkte objektreferanser. En slik situasjon oppstår når man benytter direkte objektreferanser og dermed gjør det mulig for en angriper å endre referansen via HTML eller URL [22].

Et eksempel kan være en nettbank som lar brukerne sine se kontoutskrifter ved å velge blant kontonummer i en nedtrekksliste. Nedtrekkslista inneholder bare de kontonumrene den aktuelle kunden har tilgang

til. Sett at kontoutskriften hentes basert på kontonummer uten at det kontrolleres hvilke kontoer den aktuelle kunden har tilgang til. I et slikt tilfelle vil en angriper eller en nysgjerrig bruker kunne modifisere forespørselen eller nedtrekkslista i HTML-siden og få tilgang til andre brukeres kontoutskrifter.

Fallgruven omfatter ikke bare identifikatorer som for eksempel kontonummer nevnt i eksempelet ovenfor, men også filreferanser og databasetabeller og -nøkler.

Man unngår å gå i fallgruven ved så langt det er mulig å la være å benytte direkte objektreferanser. I de tilfellene man benytter slike referanser må man kontrollere at forespørslene som benytter referansen er gyldige og føre tilgangskontroll med den aktuelle ressursen.

Cross-site request forgery (XSRF)

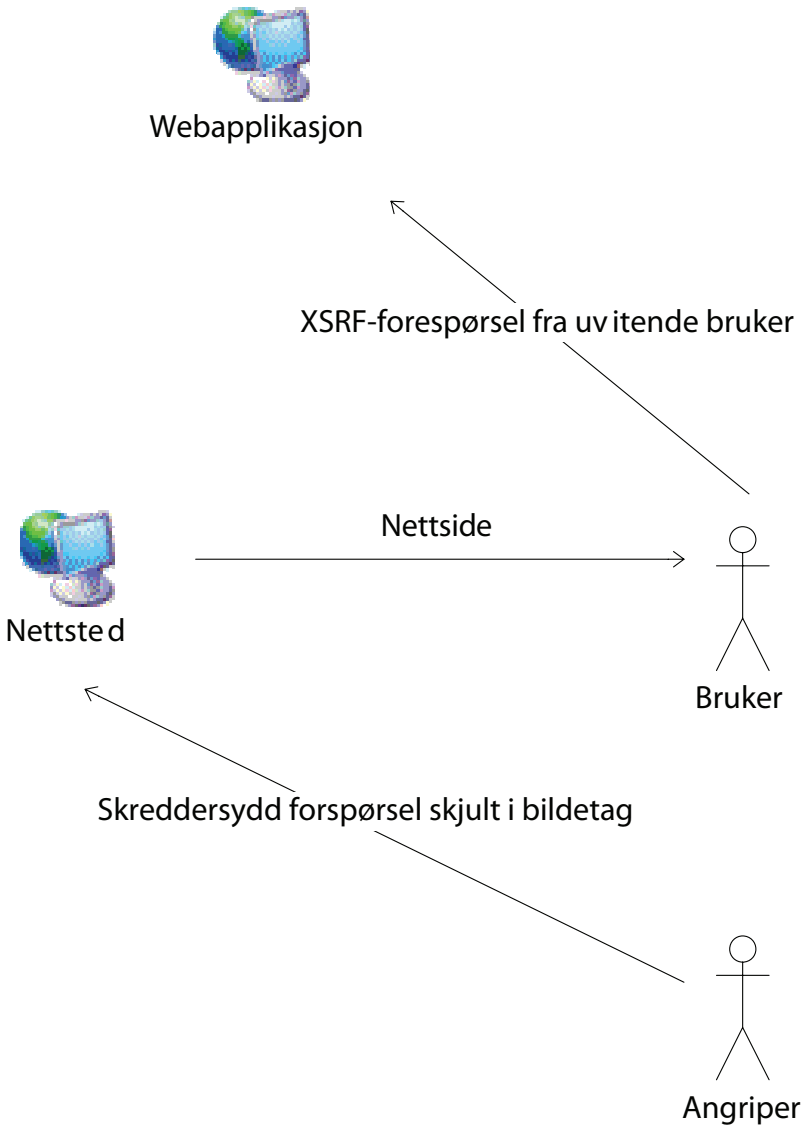
Cross-site request forgery er også kjent som «one-click attack» eller «session riding», og er et angrep som utnytter den tilliten en webapplikasjon har til en brukers nettleser [20]. En angriper som benytter XSRF lurer sine ofre til å sende forespørsler uten at de selv er klar over det. Dette utgjør et sikkerhetsproblem når ofrene er logget inn i en webapplikasjon som er åpne for dette angrepet.

Noen webapplikasjoner benytter cookies (se side 8) for å kontrollere om en bruker er innlogget, andre sjekker om forespørselen kommer fra en IP-adresse som befinner seg innenfor et gitt utvalg av IP-adresser. Uansett hvilken måte det gjøres på er hensikten at applikasjonen som mottar forespørselen skal kunne avgjøre hvilke operasjoner avsenderen har lov til å utføre.

For å ta et eksempel: Bruker A er logget inn i nettbanken sin. Samtidig har han oppe en nettside som inneholder et diskusjonsforum. Innleggene i forumet inneholder brukerdefinert HTML-kode. Angriper B har lagt inn følgende HTML-kode i sitt innlegg:

```
<img src="http://www.nettbank.no/betal?bruker=b&sum=10000">
```

Forespørselen som ligger i img-taggens src-attributt sendes når siden vises i As nettleser, og den sendes fra nettleseren til bruker A. Angrepet effektueres når nettbanken så mottar forespørselen. Nettbankapplikasjonen vil da kontrollere sesjons-id'en og oppdage at forespørselen kommer fra en bruker som er innlogget, og følgelig anta at alt er i orden. Forespørselen godtas og utføres. I dette eksempelet vil dette i praksis medføre at bruker A uten å være klar over det overfører 10 000 kroner til angriperen via sin nettbank.



Figur 3.2: XSRF

En av forutsetningene for XSRF er en webapplikasjon som angriperen kan skreddersy forespørsler til [22]. Ulempen med XSRF er derfor, sett fra angriperens ståsted, at det trengs inngående kjennskap til webapplikasjonen slik at det er mulig å skreddersy en virkningsfull forespørsel. Som vi ser av eksempelet er en annen karakteristikk ved dette angrepet at det er vanskelig for offeret å oppdage.

For å beskytte seg mot denne typen angrep er første steg å fjerne alle

XSS-sårbarheter i applikasjonen [22]. I tillegg kan man implementere et «billettsystem» for POST-forespørsler. Dette gjøres ved å sende med et skjult felt, en «billett» med en vilkårlig utvalgt verdi hver gang man sender en respons fra server til klient, for eksempel:

```
<input name="ticket" type="hidden" value="943912154">
```

Når så klienten sender inn en form eller en URL inkluderes denne verdien i forespørselen og sikrer at forespørselen kommer fra rett bruker, og man unngår XSRF.

En annen fremgangsmåte for å unngå XSRF er å benytte re-autentisering. Dette er en løsning som brukes i enkelte nettbanker, og innebærer at brukeren må bekrefte handlingen ved utføring av enkelte oppgaver. For eksempel kan dette dreie seg om re-autentisering ved overføring av penger. Hvis ikke angriperen har mulighet til å få tak i passordet eller koden som trengs for å få bekreftet handlingen, vil det ekstra steget eliminere muligheten for å få gjennomført XSRF. Re-autentisering kan skje ved at applikasjonen ber om engangskode eller brukernavn og passord. I enkelte nettbanker benytter brukerne en egen enhet som kalles passordkalkulator til å re-autentisere seg. I slike løsninger generérer enheten koden som trengs.

Utilstrekkelig feilhåndtering og informasjonslekkasjer

Angripere forsøker å skaffe seg mest mulig kunnskap om målene sine. Denne aktiviteten kalles «profiling» [21]. En måte angripere foretar profiling på er at de skaper feilsituasjoner som igjen fører til feilmeldinger. Enkelte webrammeverk kan lett konfigureres på en slik måte at detaljert, teknisk informasjon avsløres via feilmeldinger som blir sendt tilbake til brukeren når det oppstår feilsituasjoner. Dette er ofte tekniske meldinger som en normal sluttbruker ikke vil ha verken glede eller nytte av å se [11]. Informasjonen som vises kan være databasespørringer, stack traces, feilkoder eller annen informasjon som kan være til nytte for en utvikler som debugger applikasjonen [22].

Man unngår denne typen informasjonslekkasje ved å gi enkle, ikke-tekniske tilbakemeldinger til brukerne. Vanlige brukere kan oppfatte omfattende, tekniske feilmeldinger som skremmende. Derfor bør slike tilbakemeldinger unngås. I mange tilfeller ligger løsningen i korrekt konfigurasjon av rammeverk og serverprogramvare, og det er viktig å unngå at brukere og dermed også angriperne eksponeres for teknisk informasjon eller gis tilgang til debuggingsmekanismer.

Utilstrekkelig autentisering og sesjonshåndtering

Mange webapplikasjoner har innloggingsfunksjonalitet og benytter cookies for å holde rede på brukernes sesjoner. Ofte benyttes brukernavn og passord som akkreditiver for slik autentisering [11]. Ulempen med dette er at brukernavn og passord er statiske, slik at når en angriper først har skaffet seg denne informasjonen kan han/hun logge seg inn i webapplikasjonen flere ganger og når det måtte passe.

Feil i hovedautentiseringsmekanismen er ikke uvanlig, men like ofte forekommer feil i tilleggsfunksjonalitet som utlogging, «husk-meg»-funksjonalitet, passordhåndtering og oppdatering av brukerkonto [22]. Det er for eksempel viktig at sesjonen på serveren slettes eller ugyldiggjøres ved utlogging, slik at en angriper ikke kan nyttiggjøre seg av sesjonen etter at utlogging har funnet sted. Et annet tiltak som kan gjøre situasjonen vanskeligere for angripere er å gi sesjoner en viss levetid før de slettes. I tillegg må alternative autentiseringsmekanismer være like trygge som hovedmekanismen, ettersom systemets totale autentiseringsmekanisme aldri vil være sterkere enn det svakeste leddet. Slike mekanismer kan for eksempel være generering av nytt passord eller at brukeren svarer korrekt på et spørsmål han/hun har lagret definisjon på i systemet tidligere.

Datatrafikk

Kommunikasjonen mellom en webklient og en webserver passerer en infrastruktur på veien. En angriper som har tilgang til å lytte på datatrafikken vil kunne hente ut sensitiv informasjon som brukernavn og passord. Slik aktivitet kalles «packet sniffing» [11]. Det er også mulig for angripere å foreta såkalte «Man-in-the-middle»-angrep (MITM). Der «packet sniffing» er en passiv angrepsmåte, tar et MITM-angrep aktiv del i kommunikasjonen. Dette skjer ved at angriperen lurer brukeren til å opprette en forbindelse til angriperens server istedenfor det opprinnelige målet. Angriperen videresender så data i begge retninger. Dette åpner for at angriperen ikke bare kan lytte på datatrafikken, men også modifisere den [11]. Det finnes flere måter en angriper kan avlytte datatrafikk på. Dersom angriperen og offeret befinner seg på samme nett, er det mulig for angriperen å foreta såkalt ARP spoofing. ARP står for Adress Resolution Protocol [14]. Angrepet går ut på at angriperen omadresserer pakkedata ved å assosiere en IP-adresse med feil MAC-adresse, slik at pakkene blir sendt til angriperens datamaskin istedet for brukerens [11].

Dersom angriperen ikke befinner seg på samme nett som offeret, vil ikke ARP spoofing fungere. Det kan allikevel være mulig for angriperen

å lure offeret til å sende forespørsler til angriperens datamaskin [11]. DNS spoofing er en angrepsmåte hvor angriperen sender offeret uriktige DNS-adresser. Dette åpner for at angriperen kan lure offeret til å sende forespørsler til et annet mål enn hva som var intensjonen, for eksempel til angriperen selv.

For å sikre kommunikasjonskanalen mellom klient og server er det vanlig å kryptere kommunikasjonen (se 2.1.2), men også denne løsningen har svakheter [11] (se side 10).

Ondsinnet fileksekvering

Applikasjoner som henter filnavn fra parametre eller benytter filer som brukere har lastet opp, er potensielt sårbare for ondsinnet fileksekvering. Kontrolleres ikke slike potensielt svake punkter for utnyttelse, kan det være mulig for angripere å eksekvere filer med ondsinnet kode på serveren [22]. Løsningen er å sørge for at brukerne ikke får anledning til å modifisere filreferanser eller plassere skadelige filer på serveren.

Utrygg kryptert lagring

Punkt 8 på OWASPs Top ten-liste omhandler mangelfull kryptering av data som lagres i webapplikasjoner. De mest vanlige fallgruvene i denne sammenhengen er:

- Manglende kryptering av sensitive data.
- Bruk av egenutviklede krypteringsalgoritmer.
- Feil bruk av korrekte algoritmer.
- Bruk av utdaterte algoritmer (MD-5, SHA-1 med flere).
- Hardkoding av nøkler og ubeskyttet lagring av nøkler.

Utilstrekkelig tilgangskontroll for URLer

Det siste punktet på OWASPs liste omhandler mislykket eller mangelfull tilgangskontroll for URLer. Dette kan for eksempel dreie seg om webapplikasjoner som innehar URL-adresser som ikke er lenket til fra noe sted i applikasjonen, men som en angriper kan finne eller gjette seg frem til og bruke til å angripe applikasjonen via. En slik tilnærming til sikkerhet (security by obscurity) kan i beste fall gjøre det tungvint og tidkrevende for en angriper å kompromittere en webapplikasjon, men frarådes

som sikkerhetsmekanisme på grunn av den åpenbare faren for angrep. Eksempler på slik utilstrekkelig tilgangskontroll ser man for eksempel i webapplikasjoner som tilbyr brukergrensesnitt for applikasjonsadministrasjon via en skjult nettside [11]. Selv om det ikke er mulig å komme til adminsiden ved å følge lenker fra nettstedet, vil man på ingen måte være sikret mot at en angriper gjetter adressen og på den måten skaffer seg tilgang.

3.4 Sikkerhet i rike internettapplikasjoner

Som nevnt i 2.2 er rike internettapplikasjoner vanskelig å definere eksakt. Det er derfor også umulig å generalisere og fastslå hvorvidt rike internettapplikasjoner er sikre eller ei - det kommer helt an på eksakt teknologi og kontekst.

Hvilken relevans har sikkerhet for rike internettapplikasjoner? RIAer er som nevnt i 2.2 bygget opp på den måten at det kjøres kode, gjøres beregninger og kanskje også mellomlagres data på klientsiden. Dette står i kontrast til tradisjonelle webapplikasjoner hvor alt innhold ble generert på serversiden og deretter sendt tilbake til klienten, som kun hadde i oppgave å vise innholdet den fikk fra serveren. Når dette nå endres og klienten selv innehar tilstand, blir utveksling av informasjon mellom klient- og serverside annerledes enn vi er vant til. Hva har dette å si for sikkerheten? Hvordan kan en slik applikasjon takle den totale usikkerheten om hva som foregår på klientsiden og fortsatt være sikker og tilby god funksjonalitet? Hva bør utviklere gjøre for å lage trygge, rike internettapplikasjoner, og hvilke fallgruver bør de passe seg for?

Som vi ser er det en rekke spørsmål som bør besvares når det gjelder sikkerhet i denne typen applikasjoner. Ved å finne svar på disse spørsmålene og gjøre utviklere klar over hvilke sikkerhetsutfordringer som finnes, vil det være mulig å finne fram til såkalte «best practices». Ved hjelp av slike «best practices» i form av enkle råd og regler for utviklere, vil det bli lettere å lage tryggere applikasjoner uten å måtte bruke store mengder tid på å sette seg inn i hele fagfeltet. Det blir lettere å ta informerte avgjørelser i forhold til sikkerhet, og utviklerne kan selv avgjøre når en applikasjon er sikker nok. Når utviklere lager tryggere applikasjoner, vil brukerne kunne stole på og dermed bruke denne nye generasjonen webapplikasjoner.

Kapittel 4

Beskrivelse av arbeidet

4.1 Demonstrasjonsapplikasjoner

I arbeidet med å sikkerhetsteste rammeverk for utvikling av RIAer er det naturlig å ha tilgang til applikasjoner av denne typen. Det er også en fordel å kjenne så godt til applikasjonslogikken, infrastrukturen og applikasjonsarkitekturen som mulig, i tillegg til å kunne sammenlikne egenskapene til ulike rammeverk. Det har derfor vært en del av denne masteroppgaven å teste rammeverkene omtalt i 2.2.2. Å teste alle rammeverkene, teknologiene og teknikkene er utenfor hva som hadde vært mulig innenfor tidsrammene av denne masteroppgaven. Det ble derfor besluttet å velge ut to av dem og utvikle to demonstrasjonsapplikasjoner med samme funksjonalitet. Dette gjør det mulig å sammenlikne rammeverkene, samtidig som utviklingsarbeidet ikke tar opp så mye av den totale tiden at det blir vanskelig å gå i dybden med tanke på sikkerhetstematikken.

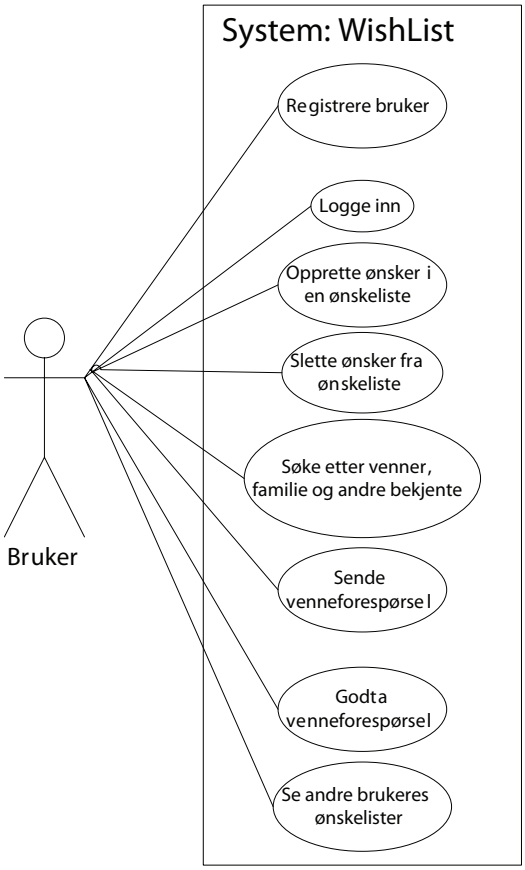
4.1.1 Funksjonalitet

Applikasjonene som ble utviklet er utformet som nettsamfunn for brukere som ønsker å utveksle ønskelister, for eksempel i forbindelse med jul, bursdager og lignende. Applikasjonene krever at brukerne registrerer seg. På registreringstidspunktet vil brukerne bli bedt om å oppgi fornavn, etternavn, e-postadresse samt velge et brukernavn og et passord. Vellykket registrering forutsetter at e-postadressen ikke finnes i systemet fra før og at brukernavnet ikke er i bruk.

Brukeren vil deretter ha anledning til å logge inn og opprette en egen ønskeliste, hvor han/hun for hvert ønske som opprettes blir bedt om å oppgi navn eller type, pris, antall og eventuell adresse til nettside som har

varen eller produktet.

Videre har brukerne anledning til å søke opp andre brukere av applikasjonen ved hjelp av navn. De vil så kunne sende venneforespørsler. Blir forespørselen godkjent av mottakeren, vil de to kunne se hverandres ønskelister.



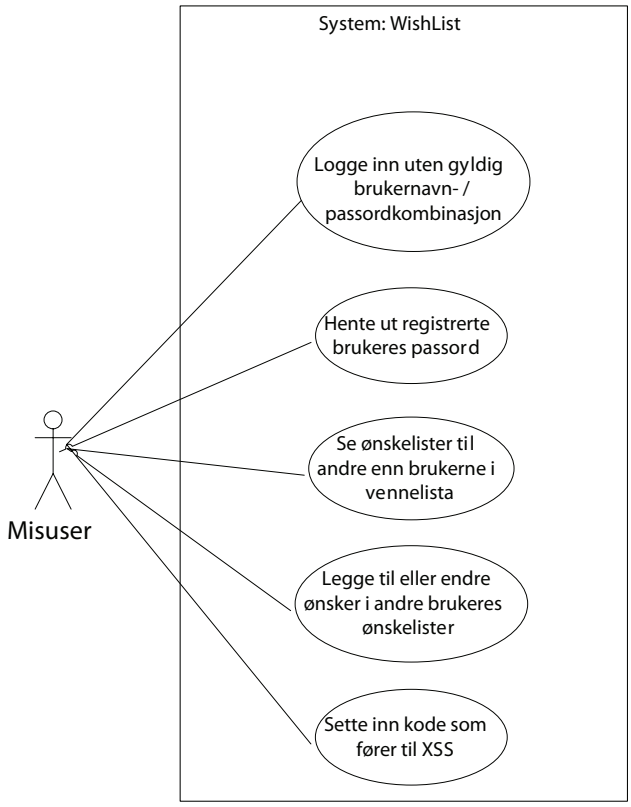
Figur 4.1: Usecase

4.1.2 Sikkerhetsaspekter

Ettersom temaet i denne oppgaven er sikkerhet, ble det lagt vekt på å utvikle en applikasjon som skulle være interessant fra et sikkerhetsperspektiv. Med dette menes at applikasjonens funksjonalitet skulle gjøre det mulig å teste for eksisterende sikkerhetssvakheter.

Viktige sikkerhetskrav til ønskelisteapplikasjonen er at det ikke skal være mulig å:

- Avsløre brukernes brukernavn og passord.
- Logge inn uten gyldig brukernavn og passord.
- Se ønskelistene til andre enn de man har i vennelista.
- Legge til eller endre ønsker i andre ønskelister enn sin egen.
- Skjule ondsinnede script eller lenker til slike script.
- Avlytte kommunikasjonen mellom klient- og serverside.



Figur 4.2: Misusecase

Det ble i utviklingen av serverapplikasjonen ikke implementert sikkerhetsmekanismer eller på annen måte gjort forsøk på å unngå å gå i kjente fallgruver. Hensikten med dette var å ha en demonstrasjonsapplikasjon som var åpen for angrep, slik at det skulle være mulig å avdekke, dokumentere og foreslå måter å unngå fallgruver på i kontekst av denne oppgaven og de teknologiene den omhandler.

Det ble i arbeidet med å teste applikasjonene delt inn i to typer sikkerhetssvakheter:

- Sårbarheter som skyldes rammeverket og som krever oppdatering av dette fra produsenten.
- Sårbarheter som skyldes programkoden eller lokal konfigurasjon.

4.1.3 Valg av rammeverk

Som nevnt i 2.2.2, finnes det en rekke rammeverk, teknikker og teknologier som muliggjør utvikling av RIAer. Ønsket i denne oppgaven var å velge ut de to mest egnede rammeverkene blant disse. Viktige argumenter ved valg av rammeverk var:

- Utbredelse. Hvor mange implementasjoner i den virkelige verden benytter dette rammeverket?
- Plattformstøtte. Hvor god plattformstøtte har teknologien?
- Ytelse. En felles karakteristikk ved rike internettapplikasjoner er at det kjøres kode på klientsiden. Hvor effektivt er rammeverket med tanke på ytelse?
- Modenhet og fremtidsutsikter. Er rammeverket helt ferskt, eller har det blitt modent gjennom flere versjoner? Hvilken fremtid har rammeverket?
- Nettleserportabilitet. Vil en applikasjon se ut og oppleves identisk i ulike nettlesere?

Følgende argumenter ligger til grunn for avgjørelsen:

- ActiveX er offisielt kun støttet i Microsoft Explorer. Teknologien er dessuten i ferd med å bli utrangert, og har slitt med sikkerhetsutfordringer.
- JavaFX er i skrivende stund i versjon 1.2. Rammeverket ble lansert i desember 2008 og fremstår som et svært ungt rammeverk. Forsøk på å finne gode eksempler på JavaFX-applikasjoner eller store implementasjoner har ikke lykkes.

- Applikasjoner laget ved hjelp av Google Web Toolkit kan i enkelte tilfeller oppleve ytelsesproblemer grunnet at de er basert på JavaScript. Hvor raskt ulike nettlesere kjører JavaScript varierer. Et av hovedargumentene for GWT er at utviklere ved hjelp av rammeverket enkelt kan skrive webapplikasjoner som støtter flere ulike nettlesere. Google selv anbefaler allikevel at man tester applikasjonen sin i forskjellige nettlesere for å verifisere dette. Miljøet en GWT-applikasjon vil kjøre i på klientsiden er ikke pluginbasert, men avhengig av hver enkelt nettlesers JavaScript-tolkning. Dette kan føre til ulik brukeropplevelse avhengig av hvilken nettleser man bruker.

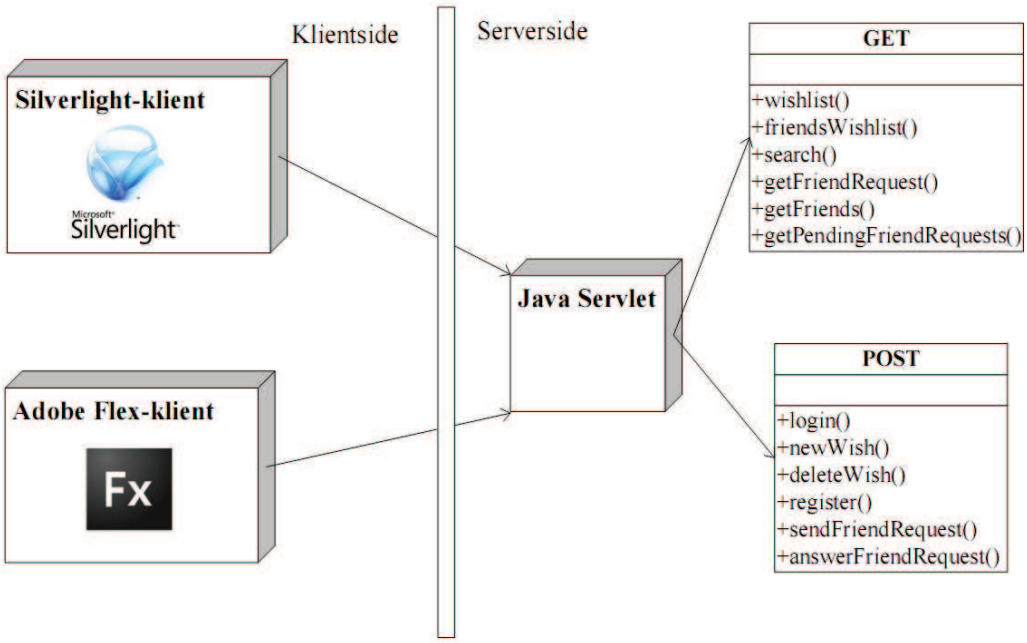
- Flex og Silverlight er moderne, men utbredte teknologier som begge kjører i et plugin-miljø. De to rammeverkene fremstår som naturlige konkurrenter.

Valget falt på Adobe Flex fordi dette er et godt egnet rammeverk ment spesifikt for formålet RIA-utvikling. Flex har modnet gjennom versjon 1 og 2 til 3.4, som i skrivende stund er den nyeste. Flex fremstår som et av de mest modne RIA-rammeverkene. Rammeverket er benyttet i en rekke omfattende implementasjoner som for eksempel Sliderocket og Scrapblog (se 2.2.1). Flex benytter Adobe Flash Player til å kjøre applikasjonene på klientsiden, og er på denne måten sikret plattformstøtte for både Linux-, Microsoft- og Mac-operativsystemer.

Til tross for noe usikker eller manglende plattformstøtte (se tabell 2.2), ble rammeverk nummer to Microsoft Silverlight. Begrunnelsen for dette valget er at Silverlight er en naturlig konkurrent til Flex, ment for utvikling av RIAer og med enkelte likhetstrekk til konkurrenten Flex. Dette veide tyngre enn plattformstøtte i denne avgjørelsen. Alternativet var Google Web Toolkit, som jo har bred plattformstøtte siden det er basert på tradisjonell HTML og JavaScript, men som er basert på noe eldre teknologi og må hanskes med forskjellige JavaScript-miljøer i ulike nettlesere.

4.1.4 Serverapplikasjon

Både Flex og Silverlight er rammeverk for å lage applikasjoner som kjører på klientsiden. For at applikasjonene skal kunne lagre, hente og oppdatere data sentralt på en server, er det nødvendig å kommunisere med en serverapplikasjon som tilbyr slik funksjonalitet. Serverapplikasjonen tilsvarer boksen med påskriften «Java Servlet» i figur 4.3. Applikasjonen kjøres på en Apache Tomcat servlet container versjon 6.0.18.



Figur 4.3: Oversikt over systemet

Serverapplikasjonen som ble utviklet benytter enkelte idéer fra Representational state transfer (REST)-arkitektur for å unngå tette forbindelser mellom klient- og serverapplikasjon. Klienten sender forespørsler til serveren ved å benytte en av de to HTTP-metodene POST og GET. Sistnevnte benyttes i tilfeller der forespørselen ikke innebærer endring av tilstand, som for eksempel ved søk eller henting av ønskeliste eller venneoversikt. I tilfeller hvor forespørselen medfører endring, så som ved innlogging, sletting av ønske eller sending av venneforespørsel, benyttes POST (se side 7 for forklaring).

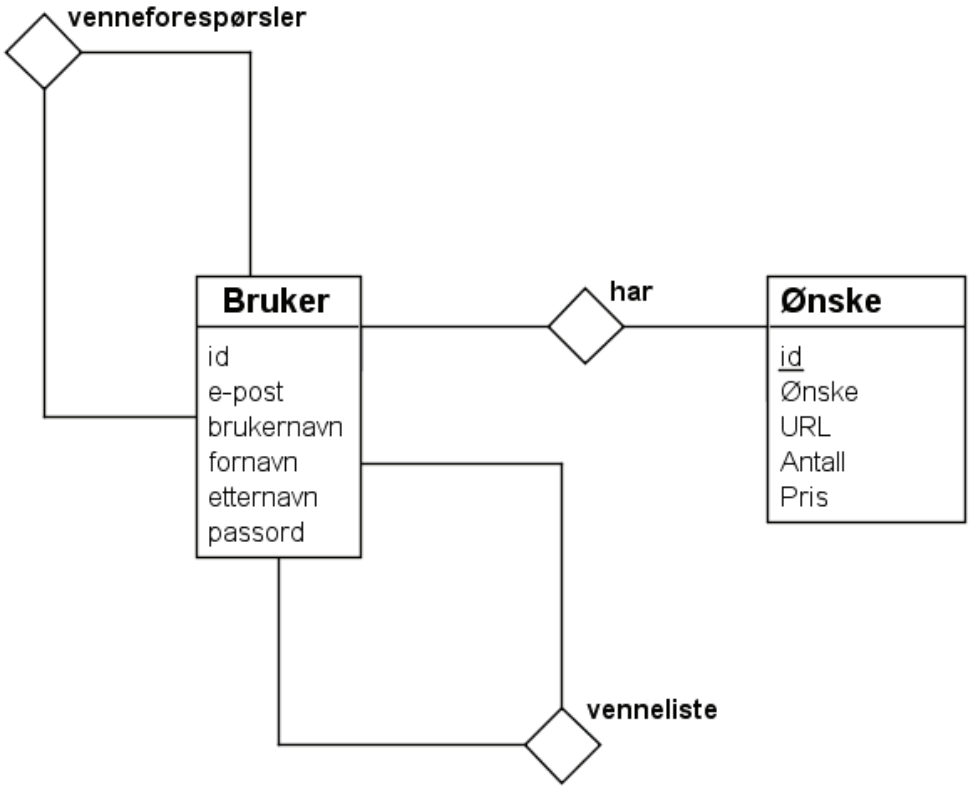
Forespørsler til applikasjonen går til et servletobjekt som delegerer forespørselen videre basert på URL.

Applikasjonen benytter en MySQL relasjonsdatabase for persistens.

Datautveksling

Både Adobe Flex og Microsoft Silverlight har flere alternativer når det gjelder utveksling av data med serversiden. Ved kommunikasjon med serversiden kan man velge å bruke RESTful webservices eller SOAP.

REST er en form for programvarearkitektur eller såkalt «design-pattern» til bruk ved utvikling av webapplikasjoner [8]. SOAP er en



Figur 4.4: Entity-Relationship diagram

protokollspesifikasjon for datautveksling med webservices.

Med Flex er det også mulig å benytte datautvekslingskomponenten «RemoteObject», som gjør det mulig å koble Java-objekter på serversiden til ActionScript-objekter på klientsiden [7]. RemoteObject har også fordel av at data komprimeres før de sendes. Dette vil medføre merkbart forbedret ytelse dersom det overføres store datamengder mellom klient og server.

I tillegg til REST og SOAP støtter Silverlight også ADO.NET Data Services.

I utviklingen av demonstrasjonssystemet i denne oppgaven var det ønskelig å benytte en form for datautveksling mellom klient- og server-

side som var støttet av både Adobe Flex og Microsoft Silverlight. Dette ville gjøre det mulig å benytte samme serversideapplikasjon til begge klientsideapplikasjonene. I tillegg var det et mål å lage enkle og forståelige applikasjoner og unngå unødig komplisert datautveksling. Det var små datamengder som skulle overføres mellom server og klient. For å imøtekomme disse kravene ble det valgt en arkitektur delvis basert på REST ved utforming av datautvekslingsarkitekturen. Hensikten var å sikre løse koblinger mellom klient og server. Respons fra serveren i form av XML kan for eksempel se slik ut:

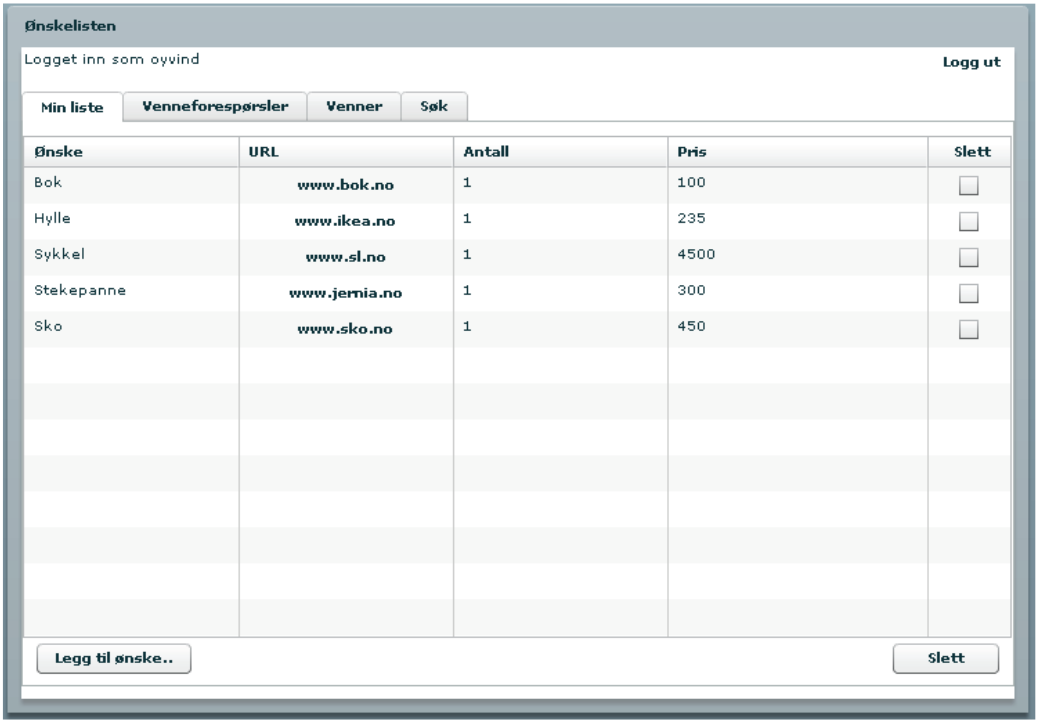
```
<?xml version="1.0" encoding="utf-8"?>
<wishes>
    <wish>
        <id>84</id>
        <name>Bok</name>
        <url>www.bok.no</url>
        <count>1</count>
        <price>100</price>
    </wish>
    <wish>
        <id>90</id>
        <name>Hylle</name>
        <url>www.ikea.no</url>
        <count>1</count>
        <price>235</price>
    </wish>
</wishes>
```

4.1.5 Demonstrasjonsapplikasjon 1: Adobe Flex

Den første applikasjonen som ble laget i demonstrasjonsøyemed benytter Adobe Flex-rammeverket versjon 3.4. Applikasjonen ble skrevet ved hjelp av Adobe Flex Builder Educational utviklingsverktøy i versjon 3. Det ble under arbeidet med å utvikle applikasjonen bevisst ikke tatt spesielle forholdsregler med tanke på sikkerhet, men rammeverkets egne komponenter for inputvalidering ble benyttet. Det vil for eksempel si at inputfeltet for e-post bare godtar gyldige e-postadresser. Komponenten har støtte for en rekke forskjellige typer feilmeldinger, og er definert i koden på følgende måte:

```
<mx:EmailValidator id="emailValidator" source="{txtEmailAdress}" property="text"
        requiredFieldError="E-postadressen mangler krøllalfa (@)"
        invalidCharError="E-postadressen inneholder et ugyldig tegn"
        invalidDomainError="Domenet i e-postadressen du har oppgitt er ikke korrekt"
        missingUsernameError="E-postadressen mangler brukernavn" required="true">
<mx:EmailValidator/>
```

Flex-applikasjonen kommuniserer med serverapplikasjonen gjennom HTTPService-objekter. Når en forespørsel skal sendes kalles den på følgende måte:



Figur 4.5: Demonstrasjonsapplikasjon: Adobe Flex

```
login_user.send();
```

Objektet «login_user» er av typen HTTPService og definert slik i koden:

```
<mx:HTTPService id="login_user" result="checkLogin(event)" method="POST"
                url="http://localhost:8080/WishList/login" useProxy="false"
                fault="handleFault(event);">

                <mx:request xmlns="">
                        <username>{txtId.text}</username>
                        <password>{txtPass.text}</password>
                </mx:request>

</mx:HTTPService>
```

Som koden viser har «login_user» et attributt, method, som angir at det dreier seg om en POST-forespørsel. Parameterne oppgis i «mx-request» taggen. «checkLogin»-metoden som angis i «result»-parameteren, kalles når forespørselen har fått respons fra webservicen.

Når forespørselen sendes går den fra Adobe Flex-applikasjonen til webservicen som er angitt i HTTPService-taggens url-attributt (se figur 4.3).

Erfaringer fra bruk av Adobe Flex

Erfaringen fra arbeidet med utvikling av demonstrasjonsapplikasjonen er at Adobe Flex er lett å komme i gang med. Takket være et godt utvalg av ferdigkomponenter som for eksempel riktekssteditor, datagrid og mediaasvspillere lar det seg gjøre å raskt sette sammen RIAer med Flex. I demonstrasjonsapplikasjonen var det spesielt datagrid-komponenten som ble brukt mye og viste seg nyttig.

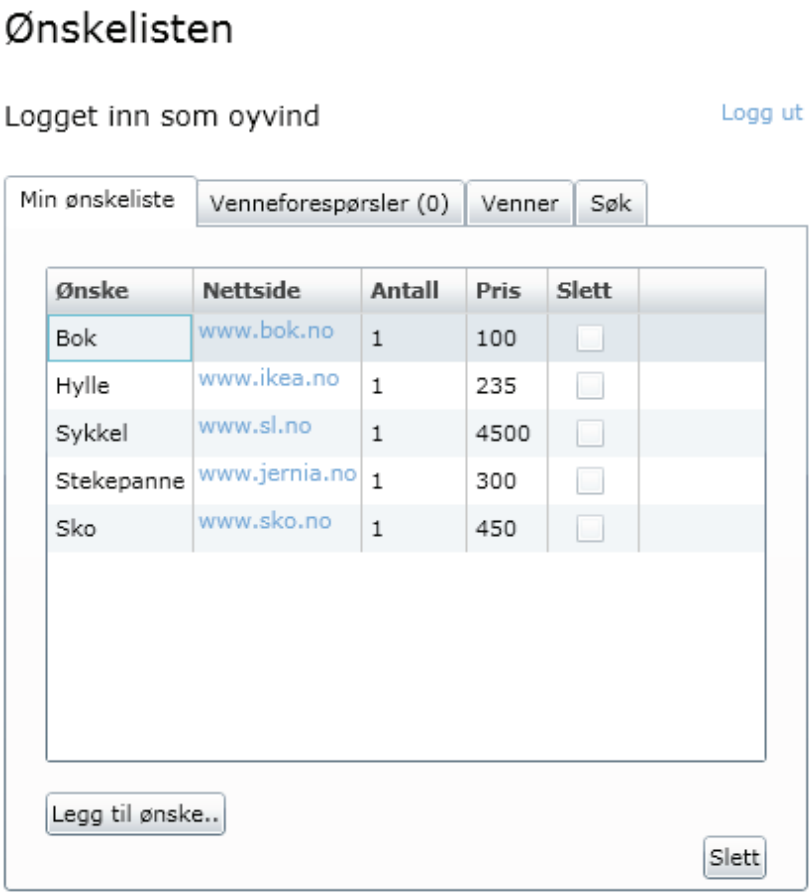
Avveiningen mellom utvikling og konfigurasjon av de medfølgende komponentene oppleves hensiktsmessig, og man slipper å finne opp hjulet på nytt hver gang man ønsker å løse et problem. Flex egner seg godt til å utvikle stilfulle brukergrensesnitt på relativt kort tid. Rammeverket kommer med et utvalg validatorkomponenter som gjør det til en overkommelig oppgave å utforme brukervennlig inputvalidering. Validatorkomponentene kan konfigureres med ønskede tilbakemeldinger avhengig av type feil.

Det er som vi ser i 4.1.5 enkelt å kalle webservicer fra Flex. Webservicen defineres i MXML-koden som vist over, og aksesseres ved å kalle en enkelt metode på objektet. Trådstyring og asynkronitet håndteres av rammeverket.

4.1.6 Demonstrasjonsapplikasjon 2: Microsoft Silverlight

Den andre applikasjonen som ble laget benytter Microsoft Silverlight versjon 2. Utviklingsverktøyet som ble brukt i utviklingsarbeidet var Microsoft Visual Studio 2008 med Silverlight-oppdateringer. I løpet av arbeidet med utviklingen av denne applikasjonen ble Silverlight versjon 3 Beta og senere versjon 3 utgitt. Arbeidet med demonstrasjonsapplikasjonen var på dette tidspunktet kommet så langt at det ble sett på som for tidskrevende og usikkert å gå over til versjon 3. Utviklingsarbeidet ble derfor fullført med Silverlight versjon 2. Microsoft har uttalt at versjon 3 er fullstendig bakoverkompatibel med versjon 2 [16]. Fordelene med versjon 3 er ifølge Microsoft blant annet disse:

- Bedre kvalitet på audio/video-streaming.
- En rekke nye brukergrensesnittkomponenter.
- Mulighet for å lage Silverlight-applikasjoner som kjører utenfor nettleseren.
- Komponenter for inputvalidering.



Figur 4.6: Demonstrasjonsapplikasjon: Microsoft Silverlight

Silverlight i versjon 2 har ikke ferdige komponenter for input-validering. Demonstrasjonsapplikasjonen innehar derfor ingen form for inputvalideringsmekanismer.

I likhet med Flex-applikasjonen benytter Silverlight-applikasjonen seg av webservices ved å sende POST og GET-forespørsler til serversiden. C#-koden som henter venneforespørsler ser slik ut:

```
private void getFriendRequests()
{
    WebClient wc = new WebClient();
    wc.OpenReadCompleted += getFriendsRequestCommandComplete;
    wc.OpenReadAsync(newRequest(urlGetFriendRequests));
}
```

Som vi ser av koden benyttes WebClient-objektet for å sende denne GET-forespørselen. openReadCompleted-attributtet på dette objektet settes til

getFriendsRequestCommandComplete, som er en metode som kalles når applikasjonen har mottatt respons fra webservicen.

Erfaringer fra bruk av Microsoft Silverlight

Ved utvikling av Silverlight-applikasjoner kan man velge blant flere programmeringsspråk (se 2.2.2). I demonstrasjonsapplikasjonen ble C# benyttet. Er man kjent med et av de programmeringsspråkene som støttes er det ikke vanskelig å komme i gang med Silverlight i og med at syntaksen vil være kjent.

Silverlight fremstår likevel som noe mer tidkrevende å komme i gang med enn Adobe Flex. Helt til å begynne med tar installasjonsprosessen noe mer tid; utviklingsverktøyet Visual Studio 2008 må installeres i tillegg til at man må installere en rekke oppdateringer for å kunne begynne å utvikle Silverlight-applikasjoner i siste tilgjengelige versjon.

Selve applikasjonsutviklingen skjer ved bruk av markup-språket XAML og det programmeringsspråket man velger å benytte, i dette tilfellet C#. Silverlight kommer med en rekke nyttige ferdigkomponenter som datagrid, grafverktøy og autocompletebox.

Når det gjelder sending av forespørsler avhenger fremgangsmåten av om det er en POST eller GET-forespørsel man ønsker å sende. Om sistnevnte er tilfelle, kan man benytte WebClient-objektet som vist i koden på side 45. Ved POST-forespørsler må man benytte HttpWebRequest-objektet, som er en noe mer omfattende prosess.

Silverlight håndterer i motsetning til Flex ikke asynkronitet automatisk, slik som Flex gjør. Man må derfor ta i mot respons fra webservicen i en egen metode. Deretter må man sende denne responsen tilbake til riktig tråd ved å kalle post()-metoden på et objekt vi kan kalle UIThread. Dette objektet må på forhånd tilordnes korrekt verdi:

```
UIThread = SynchronizationContext.Current;
```

Man må altså sende responsen fra webservicen tilbake til korrekt tråd for å kunne behandle dataene videre og oppdatere brukergrensesnittet.

For utviklere som har behov for å finjustere applikasjonen og har nytte av å detaljstyre disse prosessene kan denne måten å gjøre det på kanskje være hensiktsmessig. Dersom man på en enkel, rask og lettfattelig måte bare ønsker å hente data fra en webservice og deretter gå rett på håndtering av responsen uten å måtte ta hensyn til asynkronitet, er måten Flex løser problemet på en god del enklere å forholde seg til.

Valideringskomponenter ble først innført i versjon 3 av Silverlight. Ønsker man slik funksjonalitet og benytter versjon 2 vil dette måtte utvikles manuelt.

Alt i alt fremstår Silverlight i versjon 2 som noe mer uferdig enn Adobe Flex, men de to rammeverkene har flere grunnprinsipper til felles. Disse grunnprinsippene er for eksempel bruk av markup-basert brukergrensesnittdesign og ferdiglagede, konfigurerbare komponenter. Silverlight vil i senere versjoner absolutt kunne bli en utfordrer til Adobe Flex.

4.1.7 Forespørsler på tvers av domener

Begge de to rammeverkene Flex og Silverlight har regler for forespørsler på tvers av domener (på engelsk brukes uttrykket «cross-domain policy»). Dersom en applikasjon basert på et av disse rammeverkene ønsker å benytte data fra for eksempel en webservice, må domenet som dataene skal hentes fra gi sitt samtykke til det. Kjører for eksempel en Silverlight-applikasjon på www.silverlight.no og ønsker å hente data fra webservice som har adresse www.webservice.no, må sistnevnte gi sitt samtykke for at dette skal være mulig. Dette gjøres ved at webservicen det skal hentes data fra har en såkalt «Cross domain policy»-fil på rotnivå. Det vil si at dersom webtjenesten befinner seg på www.webservice.no/tjeneste/, må filen befinne seg på www.webservice.no.

Disse reglene gjelder ikke dersom både webtjenesten og klientsideapplikasjonen kjører på samme domene. Dersom applikasjonene kjører på forskjellig port, protokoll eller ulikt subdomene, anses dette for å være det samme som om de skulle vært på ulikt domene. Derfor vil verken www.webservice.no:8080 eller https://www.webservice.no anses som å være det samme domenet som http://www.webservice.no dersom sistnevnte kjører på port 80.

En applikasjon laget i Adobe Flex som henter data fra en webtjeneste, sender først en forespørsel og ber om å få tilsendt filen «crossdomain.xml». Finner ikke applikasjonen denne filen, vil det ikke være mulig for Flex-applikasjonen å hente data fra webtjenesten. En Silverlight-applikasjon i samme situasjon vil be om filen «clientaccesspolicy.xml». Finnes ikke den vil applikasjonen på samme måte som en Flash eller Flex-applikasjon be om «crossdomain.xml»-filen og benytte seg av denne.

En «crossdomain.xml»-fil med følgende innhold ble på plassert på rotdomenet i serverapplikasjonen:

```
<cross-domain-policy>
  <allow-access-from domain="*" />
```

```
</cross-domain-policy>
```

Filen som vises her gir alle domener adgang til å hente data fra domenet hvor filen er plassert.

4.2 Sikkerhetstesting

4.2.1 Dekompilering av klientside-kode

For å vise at det er mulig for angripere å dekompilere klientside-applikasjonene, har det vært en del av arbeidet med denne masteroppgaven å foreta slik dekompilering.

Dekompilering av Flex-kode

Flex-kode kompileres til såkalte .swf-filer. Det finnes en rekke applikasjoner tilgjengelig på internett hvor leverandørene hevder at applikasjonene er i stand til å foreta dekompilering av slike filer. Enkelte av disse verktøyene er i utgangspunktet ment for Flash og ikke Flex. Det påstås allikevel fra de som tilbyr disse verktøyene at siden Flex-applikasjoner gjøres om til ActionScript når de kompileres vil det i praksis være mulig å hente ut denne koden. Applikasjoner fra to ulike leverandører ble testet for å kontrollere hvor godt de dekompilerte .swf-filen fra demonstrasjonsapplikasjonen:

HP SwfScan

HP hevder at deres applikasjon SwfScan kan dekompilere .swf-filer, med støtte for både Actionscript 2 og 3. I tillegg hevdes det at programmet skal kunne analysere kildekoden fra dekompileringen for sikkerhetssvakheter. I dette eksperimentet var det versjon 1.0 av programmet som ble benyttet.

SourceTec Sothink SWF decompiler 4

Leverandøren av Sothink SWF decompiler hevder at programvaren i likhet med HPs programvare er i stand til å dekompilere ActionScript 3. I tillegg skal dekompilatoren kunne takle Flex-prosjekter. I forsøkene ble versjon 4.5 av denne programvaren benyttet.

Obfuskering

Det finnes flere programvareleverandører som tilbyr programvare de hevder er i stand til å kunne obfuskere ActionScript-koden som befinner seg i .swf-filer. Det ble gjennomført tester av slik programvare for å avdekke om denne typen verktøy gjør det mulig å forhindre dekompilering eller gjøre ActionScriptet i .swf-filer uforståelig.

Forsøkene gjennomføres ved at .swf-filen som inneholder demonstrasjonsapplikasjonen blir kjørt igjennom et obfuskeringsverktøy og deretter forsøkt dekompilert. Den opprinnelige koden og den dekompilerte koden vil deretter bli sammenliknet, forutsatt at obfuskert kode lar seg dekompilere. Obfuskeringsverktøyene som ble benyttet i forsøkene var Amayeta SWF Encrypt 6.0 [3] og Ambiera Irrfuscator 2.0 [4].

Dekompilering av Silverlight-kode

En nettside som inneholder en Silverlight-applikasjon har en referanse til en .xap-fil. Denne referansen finner man i nettsidens kildekode. Laster man ned .xap-filen og endrer filendelsen til .zip, vil det være mulig å åpne filen som et arkiv. Man vil da se at den inneholder blant annet .dll-filer. Disse filene inneholder den ferdigkompilerte Silverlight-koden.

Produsenten av verktøyet Reflector hevder at denne programvaren er i stand til å dekompilere .NET og Silverlight-kode. Et av forsøkene i denne oppgaven har vært å forsøke å dekompilere .dll-filene fra Silverlight-applikasjonen med dette verktøyet. Det var Reflector versjon 5.1.5 som ble benyttet i dekompileringsforsøkene.

4.2.2 Kjente sikkerhetssvakheter

I arbeidet med å teste applikasjonene for tradisjonelle sikkerhetssvakheter, ble det tatt utgangspunkt i Top Ten-listen fra OWASP [22] (se side 24).

Følgende delkapitler omhandler punktene fra OWASPs Top Ten-liste, og redegjør for hvilke fremgangsmåter som ble benyttet for å teste demonstrasjonsapplikasjonene.

Cross-Site Scripting (XSS)

Forsøket vil gå ut på å forsøke å foreta XSS ved å sende et script som input til hver av de to klientsideapplikasjonene. Dette skjer ved å fylle inn scriptet i for eksempel et av tekstfeltene i skjemaet man fyller ut for å registere et nytt ønske i ønskelista. Deretter kontrolleres det om scriptet kjøres når det vises i applikasjonens brukergrensesnitt. Følgende script-tag ble benyttet:

```
<script>alert('Hallo')</script>
```

Følgende kode ble også brukt for å teste om det var mulig å kjøre JavaScript:

javascript:alert('Hallo')

Scriptet viser et popup-vindu med teksten «Hallo» hvis det kjøres. Kjører scriptet, er applikasjonen åpen for XSS.

Injiseringssvakheter

Den vanligste injiseringssvakheten er SQL-injection, og det var denne typen injisering som ble forsøkt utført via demonstrasjonsapplikasjonene. For at angrepet skal lykkes må de injiserte SQL-kommandoene passere fritt slik at de når subsystemet, som i dette tilfellet består av en MySQL-relasjonsdatabase.

Helt konkret gikk forsøket ut på å forsøke å logge inn uten passord ved å oppgi brukernavn «oyvind@reistad.no» og injisere

hei' OR 1='1

i feltet for passord. Angrepet baserer seg på at kontrolltegn ikke filtreres og at spørringen utformes uten at parametre og selve spørringen separeres. Kontrolltegnet «'», som kommer rett etter «hei» i inputstrengen ovenfor, er ment å skulle endre kontekst slik at det som kommer etter blir en del av selve spørringen og ikke en parameter.

Usikker direkte objektreferanse

Det ble under utviklingen av de rike internettapplikasjonene ikke tatt spesielle hensyn for å unngå usikre, direkte objektreferanser. Følgelig kunne slike objektreferanser være til stede i systemet, og forsøket gikk ut på å se etter verdier og forespørsler som kunne ha denne typen direkte referanser. I tilfellet med ønskelisteapplikasjonene vil det bety å for eksempel skaffe seg urettmessig tilgang til eller endre andre brukeres ønskelister.

Cross-Site Request Forgery (XSRF)

Som nevnt i 3.3 går denne svakheten ut på at angriperen utnytter en web-applikasjons tillit til forepørsler fra en autentisert bruker. Forsøket vil gå ut på å skreddersy en forespørsel til demonstrasjonsapplikasjonene, gjemme den i for eksempel en HTML-side slik at brukeren ikke oppdager angrepet og deretter kontrollere hvorvidt applikasjonene godtar forespørselen.

Informasjonslekkasjer og ufullstendig feilhåndtering

Dette forsøket innebærer å kontrollere om noen av rammeverkene er laget eller konfigurert på en slik måte at de vil lekke informasjon når det oppstår feilsituasjoner eller unntak. Forsøket vil i praksis gå ut på å forsøke å skape feilsituasjoner slik at eventuelle tilbakemeldinger i forbindelse med feilhåndtering kan kontrolleres for informasjonslekkasje. Feilsituasjoner kan skapes ved å for eksempel returnere XML fra serversiden som ikke er velformet eller ved å sørge for at klientapplikasjonene ikke får svar fra serverapplikasjonen i det hele tatt.

Utilstrekkelig autentisering og sesjonshåndtering

Demonstrasjonsapplikasjonene benytter brukernavn og passord som akkreditiver. Sesjonshåndtering på serversiden skjer ved hjelp av det innebygde HttpSession-objektet i Java. Forsøket går ikke ut på å teste om denne mekanismen er god nok, men om bruken av den i demonstrasjonsapplikasjonen er korrekt. Konkret vil dette gå ut på å undersøke om for eksempel sesjoner går ut på tid ved en lengre periode inaktivitet og om sesjonsinformasjon slettes ved utlogging.

Det ble ikke implementert alternative autentiseringsmekanismer i demontrasjonsapplikasjonene. Forsøket vil derfor ikke ta for seg utnyttelse av denne typen «bakdører».

Usikre kommunikasjonskanaler

Kommunikasjonen som sendes mellom klient og server inneholder sensitiv informasjon som for eksempel brukernavn og passord. Det er derfor viktig at forespørslene som går mellom klient og server ikke kan avlyttes. Eksperimentet går ut på å avlytte kommunikasjonen mellom klient- og tjenerapplikasjon for å kontrollere hva det er mulig å hente ut av datatrafikken mellom de to.

I dette arbeidet benyttes programvaren Wireshark, som er laget spesielt for analyse av nettverkskommunikasjon over en rekke ulike protokoller. Dersom det viser seg at det er mulig å avlytte kommunikasjon som går over HTTP, vil det bli testet om kryptert kommunikasjon over HTTPS forhindrer avlytting.

Avgrensninger: Utelatte sikkerhetssvakheter

Følgende punkter inngår også i OWASPs Top Ten-liste, men ble utelatt fra det videre arbeidet av følgende årsaker:

- Ondsinnet fileksekvering
  Ettersom ingen av demonstrasjonsapplikasjonene tilbyr filopplasting eller har referanser til eksterne ressurser, inngår ikke testing av denne sikkerhetssvakheten i det videre arbeidet.

- Utrygg kryptert lagring
  Dette punktet ble utelatt fra det videre arbeidet i denne oppgaven fordi dette er et tema som befinner seg innenfor fagfeltet kryptografi, som er utenfor denne rapportens avgrensninger.

- Mislykket tilgangskontroll for URLer
  Dette punktet er ikke relevant i denne sammenhengen fordi begge demonstrasjonsapplikasjonene kjører i nettleserutvidelsens sandkasse i nettleseren og kun har én enkelt URL som aksesspunkt. Det er programmatisk mulig i både Flex og Silverlight-applikasjoner å hente parametre fra URL, men dette er isåfall noe utvikleren velger bevisst.

Kapittel 5

Resultater

I dette kapittelet presenteres resultatene fra arbeidet med å sikkerhetsteste systemene omtalt i 4.1.

5.1 Dekompilering av klientsidekode

5.1.1 Dekompilering av .swf-filer - Adobe Flex

Som nevnt i 4.2.1 ble det testet to forskjellige programmer som ble hevdet å skulle være istand til å dekompilere Adobe Flex kode fra .swf-filer. Dette ga følgende resultater:

HP SwfScan

Som nevnt i 4.2.1 hevder HP at SwfScan skal kunne dekompilere og analysere .swf-filer.

Dette viser seg å stemme i praksis. SwfScan var istand til å dekompilere ActionScript-koden i .swf-filen med god nøyaktighet, inkludert de opprinnelige navn på variablene (riktignok med noen siffer som prefiks). Metoden som instansierer HTTPService-objektet som muliggjør sletting av et ønske ser for eksempel slik ut i den dekompilerte koden:

```
private function _WishList_HTTPService8_i() : mx.rpc.http.mxml::HTTPService
{
    var loc0:* = new HTTPService();
    this.delete_wish = loc0;
    loc0.method = "POST";
    loc0.url = "http://localhost:8080/WishList/deletewish";
    loc0.useProxy = false;
    loc0.request = 0;
    loc0.addEventListener("result", this.__delete_wish_result);
    loc0.addEventListener("fault", this.__delete_wish_fault);
    loc0.initialized(this, "delete_wish");
```

```
    return loc0;
}
```

Programmet varsler også om potensielle sårbarheter i applikasjonen, og viser hvor i koden disse befinner seg. Det er også anledning til å be HP SwfScan lage en egen sårbarhetsrapport som oppsummerer sårbarhetene applikasjonen fant i kildekoden. Rapporten gir også en kort forklaring på hva hver sårbarhet går ut på, og hva man kan gjøre for å unngå den.

Et eksempel på en sårbarhet HP SwfScan oppdaget i demonstrasjonsapplikasjonen var bruk av nøkkelordet «password». Sårbarhetsrapporten forteller at det har blitt oppdaget treff på et nøkkelord som indikerer at man i koden avslører informasjon om en brukerkonto. Selv om nøkkelordet «password» i denne sammenheng viste seg bare å ha blitt brukt til å navngi et av inputfeltene for passord, ga dette en advarsel om at man ikke skal oppgi passord eller annen brukerkontoinformasjon i kildekoden.

SourceTec Sothink SWF decompiler

Eksperimentet med å dekompilere demonstrasjonsapplikasjonen viste at også dette programmet var i stand til å dekompilere ActionScript. En angriper vil, etter å ha dekompilert en .swf-fil med dette verktøyet, kunne se ActionScriptet som finnes i applikasjonen samt URL til alle webservicene applikasjonen benytter samt navn på parametre som sendes med. Følgende eksempel fra den dekompilerte koden er samme metode som ble vist fra dekompilering med HP SwfScan:

```
private function _WishList_HTTPService8_i() : HTTPService
{
        var _loc_1:* = new HTTPService();
        delete_wish = _loc_1;
        _loc_1.method = "POST";
        _loc_1.url = "http://localhost:8080/WishList/deletewish";
        _loc_1.useProxy = false;
        _loc_1.request = _WishList_Object10_i();
        _loc_1.addEventListener("result", __delete_wish_result);
        _loc_1.addEventListener("fault", __delete_wish_fault);
        _loc_1.initialized(this, "delete_wish");
        return _loc_1;
}// end function
```

Obfuskering

Det ble gjort forsøk på å obfuskere .swf-filen som inneholder Flex-applikasjonen, og som nevnt i 4.2.1 var det de to obfuskeringsverktøyene Amayeta SWF Encrypt 6.0 og Ambiera Irrfuscator 2.0 som ble benyttet.

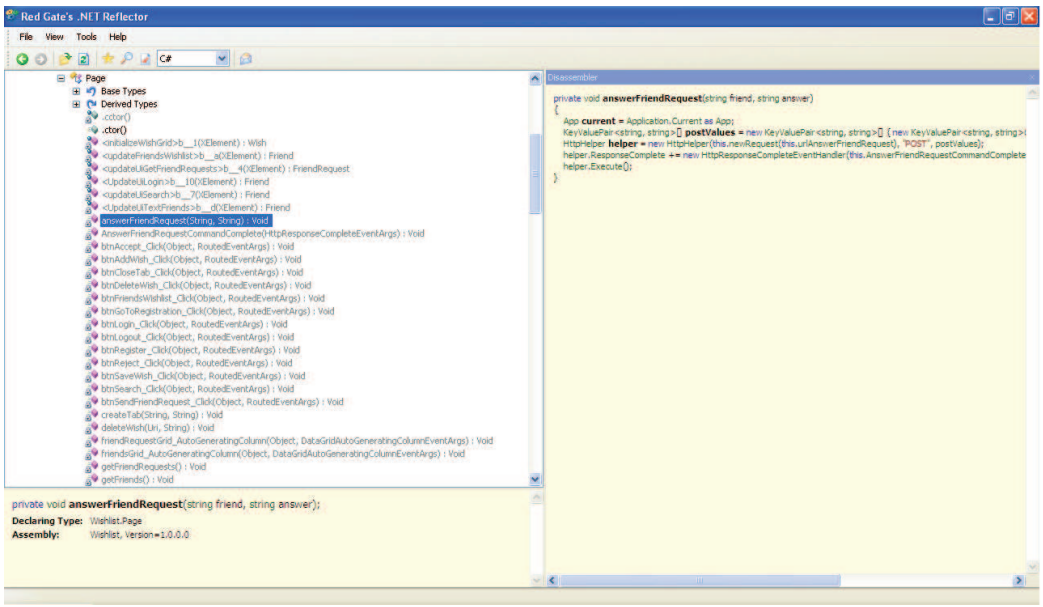
Obfuskeringsprosessen så ut til å gå greit, og applikasjonene varslet at .swf-filen ble obfuskert. Irrfuscator meldte at 382 symboler var obfuskert.

Den nye, obfuskerte filen ble lagret med opprinnelig filnavn pluss et suffiks, slik at filen som i utgangspunktet het «WishList.swf» ble lagret som «WishList_Secure.swf» eller «WishList_obfuscated». I Amayeta SWF encrypt kan man velge mellom fire forskjellige krypteringsnivåer («Encryption Setting»): «Low», «Recommended», «High» eller «Custom». Obfuskering med nivåene «High» og «Recommended» ble forsøkt.

Forsøk på å dekompilere filen etter obfuskering lykkes. Alle URL-adressene til webservicene var fortsatt synlige i klartekst i koden. Koden fremsto fortsatt som leselig og forståelig. Justering av «krypteringsnivået», som var et alternativ i Amayeta SWF Encrypt, så ikke ut til å utgjøre noen forskjell. Et utdrag fra koden hvor det vises både obfuskert og direkte dekompilert kode kan studeres i rapportens vedlegg på side 81.

5.1.2 Dekompilering av Microsoft Silverlight-applikasjon

Det ble gjort forsøk på å dekompilere Microsoft Silverlight-applikasjonen ved hjelp av verktøyet Reflector. Forsøkene var vellykkede.



Figur 5.1: Skjermbilde fra dekompilering med Reflector

Dekompilatoren viser dll-filer og hvilke klasser de inneholder. Etter dekompileringen vises variabler man definerte i den opprinnelige applikasjonen med de korrekte (altså de opprinnelige) variabelnavnene. Alle

metoder i applikasjonen vises. Koden i valgt metode vises i feltet til høyre (se figur 5.1), og tilsvarer den opprinnelige koden.

Oppsummering

Det viser seg å være ukomplisert å dekompilere både Adobe Flex og Microsoft Silverlight-applikasjoner. De to Flex-dekompilatorene gir ikke nøyaktig samme resultat, men resultatene ser ut til å ha store likhetstrekk. Obfuskering av klientsidekode så ut til å ha liten eller ingen hensikt. Det tas forbehold om at det kan finnes bedre programvare for dette formålet enn den som ble testet i denne oppgaven, men resultatet av forsøkene viser at den programvaren som ble testet her utgjør liten eller ingen forskjell.

Resultatet er som forventet. Følgene av at det er mulig å dekompilere klientsidekode er at man som utvikler må være klar over at ingenting av det som står i klientsidekoden er utilgjengelig for en eventuell angriper, selv ikke en mindre avansert angriper. Denne koden må altså skrives på en slik måte at den like gjerne kunne blitt lagt ut på internett i klartekst, ettersom den i realiteten er det allerede så snart den dekompilerbare applikasjonen er publisert. Det er viktig å ikke hardkode brukernavn, passord eller annen sensitiv informasjon i klientsidekoden. Selv om dette kan virke innlysende begår mange utviklere denne feilen [21].

Det finnes allikevel tiltak man kan gjennomføre dersom man ønsker å begrense tilgang til klientsidekoden noe. Det kan skje ved å først gi tilgang til dekompilerbar kode etter at brukeren har logget inn. I for eksempel en bedriftsintern applikasjon gjort tilgjengelig over internett vil da spredning av den dekompilerbare kildekoden begrenses til datamaskinene til de som er autentiserte brukere. Teknisk løses dette ved å sende den dekompilerbare kildekoden som respons på en vellykket innlogging.

5.2 Kjente sikkerhetssvakheter

I dette avsnittet presenteres resultatene fra forsøk der demonstrasjonsapplikasjonene testes om de er sårbare for et utvalg tradisjonelle sikkerhetssvakheter. For hver av de kjente sikkerhetssvakhetene vil det også bli ført en diskusjon rundt hvordan problemet oppstår og hva man kan gjøre for å unngå den aktuelle fallgruven.

5.2.1 Cross-Site Scripting

XSS-svakheten er en av de vanligste fallgruvene å gå i når det gjelder tradisjonelle webapplikasjoner [11]. Det viser seg at det er mulig å gå i denne fallgruven også ved utvikling av RIAer.

Adobe Flex

Adobe Flex-dokumentasjonen inneholder følgende advarsel mot XSS:

> «Developers often pass URL values to the navigateToURL() function that were obtained from external sources such as FlashVars. Attackers may try to manipulate these external sources to perform attacks such as cross-site scripting. Therefore, developers should validate all URLs before passing them to this function.»

Metoden navigateToURL() benyttes som navnet indikerer til å navigere til en URL, også ekstern, og her kan det som Adobes dokumentasjon antyder utføres XSS hvis man ikke på forhånd validerer argumentene man ønsker å sende inn til metoden. Forsøkene med demonstrasjonsapplikasjonen viser at det er mulig å kjøre JavaScript ved å lagre for eksempel følgende som URL for et nytt ønske i demonstrasjonsapplikasjonen:

```
javascript:alert("Hei")
```

Scriptet vil kjøre når lenken senere vises i ønskelista og brukeren klikker på den.

Dette viser som dokumentasjonen påpeker at input til navigateToUrl()-funksjonen må valideres dersom man ønsker å unngå å åpne for XSS. Hvordan inndataene bør valideres avhenger ifølge den samme dokumentasjonen av hvordan URLen skal brukes. Tillater man for eksempel «javascript:»-kall som vist over, blir applikasjonen sårbar for XSS.

Det ble i august 2009 også oppdaget en XSS-svakhet i selve Flex-rammeverket [5], nærmere bestemt i et JavaScript i malen for HTML-siden som omgir en Flex-applikasjon. Sårbarheten finnes i filen «index.template.html». I Flex kan man angi hvilken versjon av Flash Player som kreves for å kunne vise applikasjonen. Dette angrepet forutsetter at offeret har en eldre versjon av Flash Player enn den som kreves av den aktuelle Flex-applikasjonen. Da vil følgende kodelinje kjøres:

```
var MMredirectURL = window.location;
```

Som vi ser tilordnes variabelen «MMredirectURL» den brukerkontrollerte verdien «window.location». Variabelen blir senere skrevet direkte til HTML-siden ved hjelp av en «document.write»-kommando, og det er denne direkte bruken av brukerstyrt input som muliggjør XSS.

Leverandøren Adobe har publisert en teknisk artikkel på internett [12] som forklarer hvordan feilen kan rettes. Man kan enten oppgradere til Flex SDK 3.4 eller nyere, oppdatere HTML-malen eller endre følgende kodelinje i HTML-filen:

```
var MMredirectURL = window.location;
```

til

```
var MMredirectURL = encodeURI(window.location);
```

Microsoft Silverlight

Silverlight-applikasjoner er i likhet med Flex-applikasjoner omgitt av en HTML-side når de vises i nettleseren. HTML-siden inneholder da en referanse til Silverlight-applikasjonen, som befinner seg i en .xap-fil. Forsøk viser at det er mulig å inkludere JavaScript i denne HTML-siden via en Silverlight-applikasjon. Følgende C#-kode viser hvordan dette kan gjøres i Silverlight:

```
String javascript = "alert('Hei')";
script = HtmlPage.Document.CreateElement("script");
script.SetAttribute("type", "text/javascript");
script.SetProperty("text", javascript);
HtmlPage.Document.DocumentElement.AppendChild(script);
```

Forsøk på å få Silverlight-applikasjonen til å kjøre JavaScript direkte uten å måtte gå veien om verts-HTML-siden lykkes ikke. Dette ble testet ved å forsøke å få script til å kjøre i forskjellige Silverlight 2-komponenter som for eksempel label, TextBlock og HyperlinkButton.

Som det fremgår av kodesnutten over er dette noe man aktivt må åpne for og ikke en typisk «fallgruve» man ofte vil risikere å gå i. Det er allikvel viktig å understreke at i de tilfellene man endrer vertssiden er avhengig av å håndtere kontrolltegn korrekt dersom man ønsker å unngå script- og HTML-injisering.

Oppsummering

Konklusjonen er at Adobe Flex kan være utsatt for XSS dersom man unnlater å validere inndata ved bruk av for eksempel navigateToUrl()-funksjonen. Forutsetningen for et vellykket XSS-angrep via Flex er at

brukeren aktivt klikker på linken XSS-angrepet er basert på. Microsoft Silverlight er ikke utsatt for XSS dersom man ikke bevisst gjør det mulig for brukeren å endre HTML-siden som omgir Silverlight-applikasjonen.

5.2.2 SQL-injection

Dette eksperimentet gikk ut på å injisere SQL-kode og på den måten endre spørringer mot databasen. Først ble brukernavn «oyvind@reistad.no» tastet inn i feltet for brukernavn. Deretter ble følgende tekst fylt inn i feltet for passord:

hei' OR 1='1

Det var forventet at det ville være mulig å foreta SQL-injection i applikasjonene. Det ble under utviklingsarbeidet bevisst ikke tatt spesielle hensyn for å unngå å gå i denne typen fallgruve. Testingen av demonstrasjonsapplikasjonene viste at det var mulig å foreta SQL-injection via både Flex- og Silverlight-applikasjonen samt direkte mot webservicen.

Hvordan unngå SQL-injection

For å unngå denne typen angrep må kontrolltegn håndteres korrekt og på riktig sted i systemet [11]. Å validere input i klientsideapplikasjonene ville ført til at SQL-injection ble umulig å gjøre direkte via brukergrensesnittet. Dette er forsøkt i Flex-applikasjonen, for eksempel i feltet hvor brukeren oppgir e-postadresse ved innlogging. I det aktuelle feltet tillates ikke de spesialtegn som kreves for å få til SQL-injection. Det er allikevel viktig å være klar over at selv med en slik løsning vil angripere kunne foreta SQL-injection. Angriperen kan for eksempel:

- Dekompilere og endre klientsideapplikasjonen slik at valideringen fjernes.
- Benytte verktøy som gjør det mulig å endre forespørsler som sendes fra klientsideapplikasjonen ved å for eksempel la alle utgående forespørsler passere gjennom en lokal proxyserver, hvor forespørslene kan endres.
- Sende forespørsler direkte mot webservice og på den måten unngå klientsideapplikasjonens valideringsmekanismer.

Det finnes med andre ord ingen form for validering man kan gjøre på klientsiden som gjør det umulig for en angriper å foreta SQL-injection. Dette underbygger påstanden om at det befinner seg en «usynlig sikkerhetsbarriere» mellom klient- og serversiden [11]. Data som først har passert denne barrieren og senere sendes tilbake til serveren, må valideres. Siden en angriper har mulighet til å ta kontroll over og modifisere alle data som sendes til serveren, må man på serversiden ta høyde for alle eventualiteter når det gjelder input fra klientsiden. Dette gjelder all input til serverapplikasjonen, herunder URL-adresser, POST-data og cookieverdier.

Det er ikke trivielt å unngå SQL-injection. Som nevnt i 3.3 finnes det ulike måter å løse problemet på. En måte å er å «escape» alle kontrolltegn databasesystemet støtter, det vil si å få spesialtegn i input fra brukeren til å miste sin betydning ved å fortelle subsystemet at tegnene ikke skal fungere som kontrolltegn. Denne løsningen forutsetter at man finner ut i databasesystemets dokumentasjon hvilke kontrolltegn som støttes. Det er ikke anbefalt å forsøke å unngå SQL-injection ved å fjerne kontrolltegn fra brukerinput. Årsaken til at dette i en del sammenhenger vil være uheldig er at det iblant kan være nyttig at applikasjonen er i stand til å «escape» kontrolltegn slik at de kan lagres i databasen. Et eksempel er hvis en bruker med etternavn «O'Connor» ønsker å registrere etternavnet sitt i applikasjonen. En applikasjon som bare fjerner spesialtegn vil ikke være i stand til å lagre et slikt etternavn korrekt. I de tilfellene hvor inputdataene uansett ikke skal kunne inneholde spesialtegn og brukerne allikevel sender dem inn, er det ingen grunn til å forsøke å fjerne spesialtegnene. Årsaken er at det ikke har noen hensikt å hjelpe en eventuell angriper med å lage gyldig input [11]. I en slik situasjon bør brukeren få en feilmelding som indikerer at inputdataene inneholder ugyldige tegn.

Prepared statements

Et annet alternativ for å unngå SQL-injection er å benytte «prepared statements», en metode for å kommunisere med databasehåndteringssystemet som støtter separering av spørring og parametre.

Velger man å benytte sistnevnte metode er det viktig å være klar over at kontrolltegnet «%» ikke filtreres ut. Dette kontrolltegnet fungerer som jokertegn («wildcard» på engelsk). Forutsatt at spørringen benytter «LIKE» som sammenlikningsoperator istedenfor likhetstegn, kan angriperen sende inn jokertegn for både brukernavn og passord. Dette vil resultere i følgende spørring:

```
SELECT * FROM users WHERE email LIKE '%' AND password = '%'
```

Spørringen vil returnere den første brukeren i tabellen «users», som ofte vil være administratorbrukeren. Dette er svært uheldig, men kan enkelt unngås ved å benytte likhetstegn som sammenlikningsoperator istedenfor «LIKE» eller ved å filtrere ut jokertegn. Følgende kodeutdrag viser bruk av «prepared statements» og korrekt bruk av likhetstegn som sammenlikningsoperator:

```
String param1 = email;
String param2 = password;
String query = "SELECT * FROM users WHERE email = ? AND password = ?";

PreparedStatement ps = con.prepareStatement(query);
ps.setString(1, param1);
ps.setString(2, param2);
return ps.executeQuery();
```

5.2.3 Usikker direkte objektreferanse

Som nevnt på side 51, ble det gjort forsøk på å avsløre direkte objektreferanser i demonstrasjonsapplikasjonene. Forsøk på å finne slike direkte referanser via klientsideapplikasjonene lykkes ikke, men ved å sende forespørsler direkte mot webservicen i serverapplikasjonen lot det seg gjøre å utnytte denne typen svakhet. Forutsetningen er at man har logget inn i demonstrasjonsapplikasjonen og benytter den samme nettleseren som man har logget inn med når man sender forespørsler til serveren. Angrepet forutsetter i tillegg følgende:

- Angriperen har registrert konto i systemet og logget inn.
- Angriperen har dekompilert eller gjettet seg fram til URL til webservicene og aktuelle parametre.

Når disse forutsetningene var til stede ble det forsøkt å sende følgende forespørsel til webservicen som henter ønskelister:

http://localhost:8080/WishList/friendsWishlist?friend=29

Merk at brukeren som i dette tilfellet er innlogget ikke er «venn» med brukeren som angis i parameteren (29), slik at den innloggede brukeren egentlig ikke skal ha mulighet til å se den andre brukerens ønskeliste.

Forespørselen ga følgende XML tilbake:

```
<?xml version="1.0" encoding="utf-8"?>
<wishes>
        <wish>
                <id>29</id>
                <name>Sykkel</name>
                <url>http://www.sykkel.no</url>
                <count>1</count>
                <price>4500</price>
        </wish>
</wishes>
```

Som vi ser sendes ønskelisten til brukeren med id 29 som svar, og avslører at denne brukeren ønsker seg en sykkel. Dette viser at systemet har en usikker direkte objektreferanse som gjør det mulig å skaffe seg urettmessig tilgang til å se andre brukeres ønskelister. Den samme feilen gjelder for koden som muliggjør sletting av ønsker, som vil si at det er mulig å utnytte svakheten til å slette andre brukeres ønsker. Årsaken finner vi i serverapplikasjonens manglende kontroll over hvorvidt forespørselen er legitim eller ei.

Hvordan unngå direkte objektreferanser

Sårbarheten ville i tilfellet i eksempelet vært unngått dersom serverapplikasjonen hadde kontrollert om angriperen var venn med brukeren med id 29 og nektet tilgang hvis så ikke var tilfelle. Dette viser at det må føres tilgangskontroll på serversiden for å unngå at noen får tilgang til data de ikke har rett til å se. Java-koden på serveren kontrollerer ikke om brukeren har tilgang:

```
User user = (User)session.getAttribute("user");
if(user != null){
   Friend friend = new Friend(Integer.parseInt(request.getParameter("friend")));
   request.setAttribute("friendsWishlist", friend.getWishlist());
}
```

Problemet kan unngås ved å kontrollere tilgang før data returneres. Det kan for eksempel gjøres på følgende måte:

```
User user = (User)session.getAttribute("user");
if(user != null){
   Friend friend = new Friend(Integer.parseInt(request.getParameter("friend")));
   if(user.isFriendOf(friend.getId)){
      request.setAttribute("friendsWishlist", friend.getWishlist());
   }
}
```

Som vi ser er det i koden over lagt til en linje hvor det blir kontrollert om man faktisk har tilgang til den forespurte ønskelisten. Dette fjerner muligheten for å se andre ønskelister enn de man skal ha tilgang til.

I enkelte situasjoner er det av effektivitetsårsaker en god løsning å kontrollere input fra brukeren i klientsidekoden før man sender

forespørselen videre til serverapplikasjonen. Grunnen til det er at man unngår tidstapet ved å måtte ta en ekstra rundtur til serveren for å validere inndata. En slik løsning forutsetter at det fortsatt er implementert tilgangskontroll på serversiden i tilfelle en evenutell angriper skulle velge å sende forespørslene direkte til webservicen.

5.2.4 Cross-site request forgery (XSRF)

Det lykkes ikke å foreta XSRF-angrep direkte mot verken Flex eller Silverlight-applikasjonen, da det ikke så ut til å la seg gjøre å skreddersy forespørsler for applikasjonene. Angrepet ble deretter forsøkt utført direkte mot webservicene ved å åpne følgende HTML-side:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
        <body>
                <iframe src="iframe.html" width="0" height="0" frameborder="0"/>
        </body>
</html>
```

I denne HMTL-koden refereres det til iframe.html, som ser ut som følger:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
      <body onload="document.forms[0].submit()">
        <form method="POST" action="http://localhost:8080/WishList/deletewish">
                <input name="wish" value="90" type="text"/>
                <input type="submit" id="send"/>
        </form>
      </body>
</html>
```

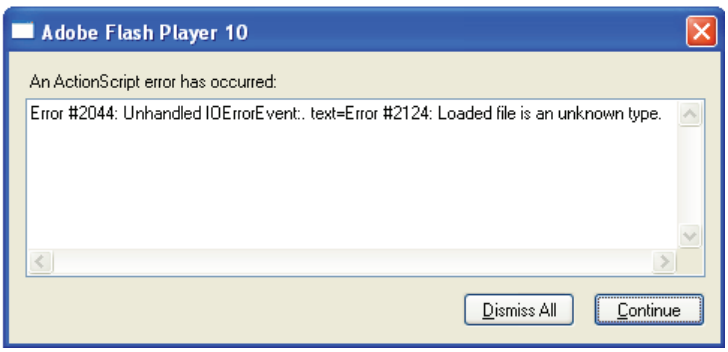
Som vi ser av koden til iframe.html, sendes en POST-forespørsel med parameter wish=90 når siden lastes. Ettersom en iframe er referert, men befinner seg i en annen HTML-fil, ser ikke brukeren denne. Det tar svært kort tid å sende forespørselen, noe som bidrar til å gjøre det vanskeligere for brukeren å oppdage angrepet [11]. Forutsetningen for at dette angrepet skal fungere er at offeret er innlogget i webapplikasjonen angrepet rettes mot, og at «angrepskoden» kjøres i samme nettleser som webapplikasjonen som er målet. Under disse forutsetningene var angrepet som er beskrevet her vellykket og førte til sletting av ønske med id=90.

En mulig løsning på dette problemet er å implementere et «billett-system» for forespørsler [11][22]. Løsningen går ut på å generére en vilkårlig tekst på serversiden og så sende med denne verdien til klientapplikasjonen. Hver gang klientapplikasjonen sender en forespørsel om å for eksempel slette et ønske i ønskelisten eller sende en venneforespørsel, sendes «billetten», den vilkårlig genererte verdien, med som en del av forespørselen. Man kontrollerer så på serversiden om verdien stemmer

overens med den som ble delt ut tidligere. Gjør den det, kan forespørselen godtas, og man er sikret at forespørselen faktisk har sitt opphav i fra klientapplikasjonen. Det vil ikke være mulig for angripere å gjette verdien av «billetten» og dermed vil muligheten for å gjennomføre XSRF-angrep være eliminert. Merk at denne sikkerhetsmekanismen kun er nødvendig å implementere for handlinger som medfører endring på serveren, altså de forespørslene som sendes som POST-forespørsel. Et eksempel på en slik handling er sletting av ønsker i ønskelisten. Søk og visning av ønskelister, derimot, medfører ingen varige endringer på serveren og kan således benytte GET og trenger ikke XSRF-beskyttelse.

5.2.5 Utilstrekkelig feilhåndtering og informasjonslekkasjer

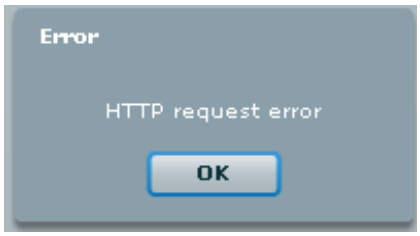
Forsøkene med å fremprovosere feilsituasjoner i applikasjonene viste at systemene under spesielle forutsetninger viser brukeren tekniske feilmeldinger og informasjon om feilsituasjonen. Det er blant annet mulig å benytte en debuggingsversjon av Flash Player som viser hvilken type feil som har oppstått (se figur 5.2).



Figur 5.2: Skjermbilde av feilmelding fra Flash Player

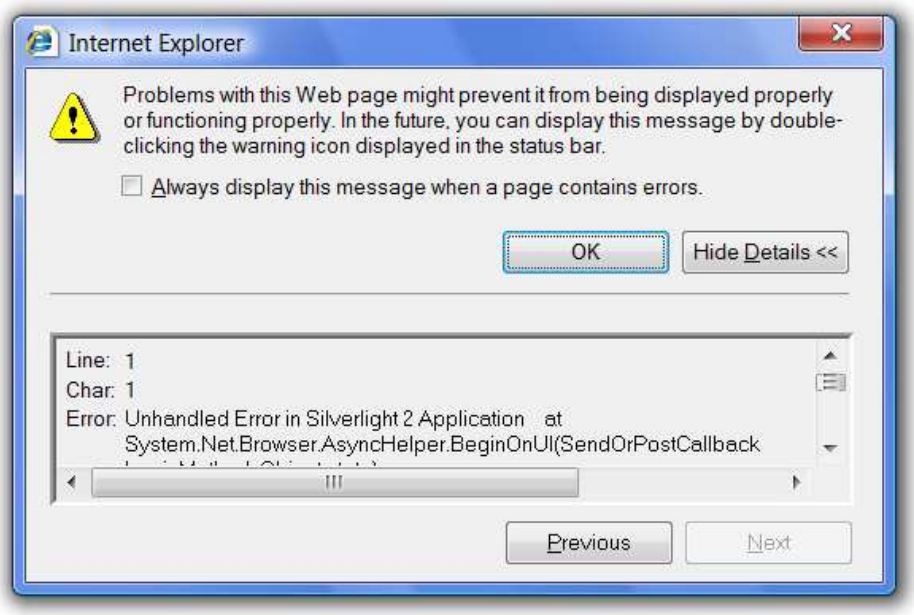
Feilmeldinger fra selve Flex-rammeverket er korte og konsise og gir brukeren en pekepinn på hva som gikk feil.

Det ble forsøkt å fremprovosere feilmeldinger også i Microsoft Silverlight. I motsetning til Flex, ser ikke Silverlight-rammeverket ut til å ha innebygde mekanismer for feilhåndtering med tilhørende feilmeldinger. Når for eksempel webservicen som kalles er utilgjengelig vil det i Flex dukke opp en feilmelding som indikerer dette (se figur 5.3). Når samme situasjon oppstår i Silverlight vil det ikke vises noen tydelig feilmelding som indikerer hva som skjer, men nettleseren (i dette tilfellet Internet Explorer)



Figur 5.3: Skjermbilde av feilmelding i Adobe Flex

viser et ikon i form av en varseltrekant nede på statuslinjen. Trykker man på ikonet vil man få se feilmeldingen som vises i figur 5.4. Dette tyder på at eventuell feilhåndtering i Silverlight 2 må programmeres manuelt dersom man ønsker å håndtere spesielle situasjoner.



Figur 5.4: Skjermbilde av feilmelding i Internet Explorer

Som vi ser viser begge rammeverkene under visse forhold tekniske feilmeldinger til brukerne. Allikevel er det verdt å merke seg at dette kun dreier seg om teknisk informasjon om klientsidesystemene. Vi har sett at kode som befinner seg på klientsiden er mulig for en angriper å skaffe seg tilgang til. Derfor vil ikke feilmeldinger som inneholder teknisk informasjon utgjøre et sikkerhetsproblem såfremt den tekniske

informasjonen kun dreier seg om klientsidearkitektur. Avslører derimot klientsidekoden tekniske detaljer om serversiden, vil det kunne oppstå informasjonslekkasjer.

I retningslinjene fra OWASP foreslås det at man i et team som gjennomfører et utviklingsprosjekt bør være enige om hvordan feilhåndtering skal gjøres, slik at det skjer på samme måte i hele applikasjonen [22]. Det anbefales også å sterkt begrense eller unngå detaljerte feilmeldinger, som er verdt å tenke over dersom man velger å håndtere feilmeldingene selv istedenfor å la rammeverket gjøre det. Dersom man i tillegg til å ha et sikkert system ønsker å ivareta god brukervennlighet, vil den beste løsningen være å gi brukere korte og informative feilmeldinger i de tilfellene hvor rammeverket ikke gjør det automatisk.

Det er også verdt å merke seg at problemstillingen med avslørende feilmeldinger er mest sentral for serverapplikasjoner og webserverprogramvaren man velger å benytte. Vel så ofte som feil i selve webserveren forekommer feilkonfigurering som åpner for detaljerte, tekniske tilbakemeldinger til glede og nytte for angripere.

5.2.6 Utilstrekkelig autentisering og sesjonshåndtering

For å kunne se detaljert informasjon om cookies på klientsiden ble det benyttet Mozilla Firefox nettleser versjon 3.0.14 med Web Developer utvidelse. Forsøk med demonstrasjonsapplikasjonene viste at det fantes feil ved bruken av autentiseringsmekanismen. Konkret viste det seg at sesjonen ikke ble avsluttet selv om brukerne klikket på «Logg ut»-lenken. Etter å ha klikket på lenken, viser klientsideapplikasjonen kun et innloggingsfelt. I denne situasjonen er det meningen at det ikke skal være mulig å benytte annen funksjonalitet enn å logge inn på nytt. Allikevel avslørte testene at det fortsatt mulig å sende forespørsler direkte mot webservicene, og undersøkelser viste at cookien på klientsiden fortsatt var intakt. Dette tyder på at sesjonen fortsatt er aktiv. Feilen gjaldt begge klientsideapplikasjonene.

En utloggingsfunksjon må kalle serversiden slik at denne blir informert om at brukeren ønsker å logge ut og at sesjonen skal opphøre. Sesjonen avsluttes i serverapplikasjonen ved hjelp av følgende Javakode:

```
session.invalidate();
```

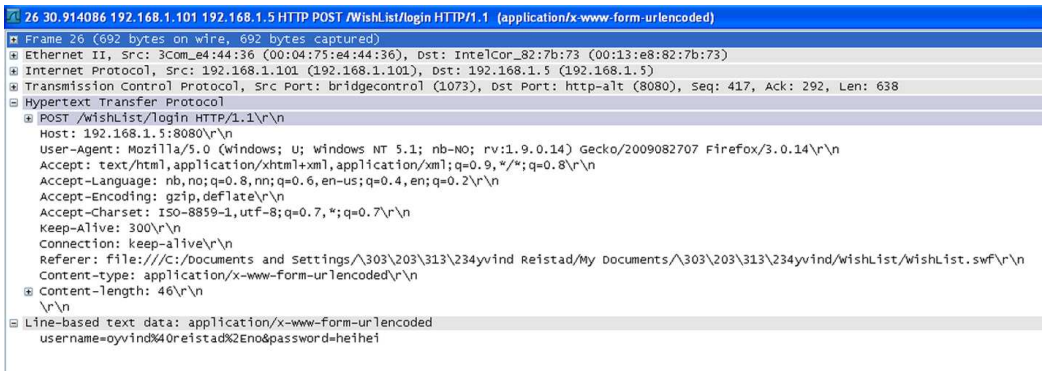
Dette fører til at sesjonen ikke lenger er gyldig, og forutsatt at serversiden er korrekt implementert vil brukeren måtte logge inn på nytt for igjen å få tilgang til webservicene.

Å avslutte sesjoner på serversiden og å slette eller ugyldiggjøre cookien på klientsiden anbefales dersom man ønsker en sikker webapplikasjon. Skjer ikke dette vil det for det første være enkelt for en angriper med fysisk tilgang til datamaskinen som har vært brukt å fortsette å benytte webservicene, ettersom nettleseren på den aktuelle datamaskinen fortsatt vil være autentisert mot serveren. For det andre vil det gjøre at tiden en bruker er autentisert mot webapplikasjonen øker, noe som vil være en fordel for angripere.

5.2.7 Usikre kommunikasjonskanaler

Forsøket på å avlytte kommunikasjonen hadde til hensikt å kontrollere om det var mulig å hente ut sensitiv informasjon som for eksempel brukernavn og passord ved å lytte på kommunikasjon mellom serverapplikasjonen og de to klientsideapplikasjonene. Verktøyet Wireshark gjør det mulig å lytte på trafikk over forskjellige nettverksprotokoller og ble benyttet i dette arbeidet.

Ettersom tekniske begrensninger gjør det vanskelig å lytte på lokal trafikk (localhost) ble klient- og serversideapplikasjonen kjørt på hver sin datamaskin. Avlyttingsverktøyet Wireshark kjørte på serversidedatamaskinen.



Figur 5.5: Avlytting av HTTP med Wireshark

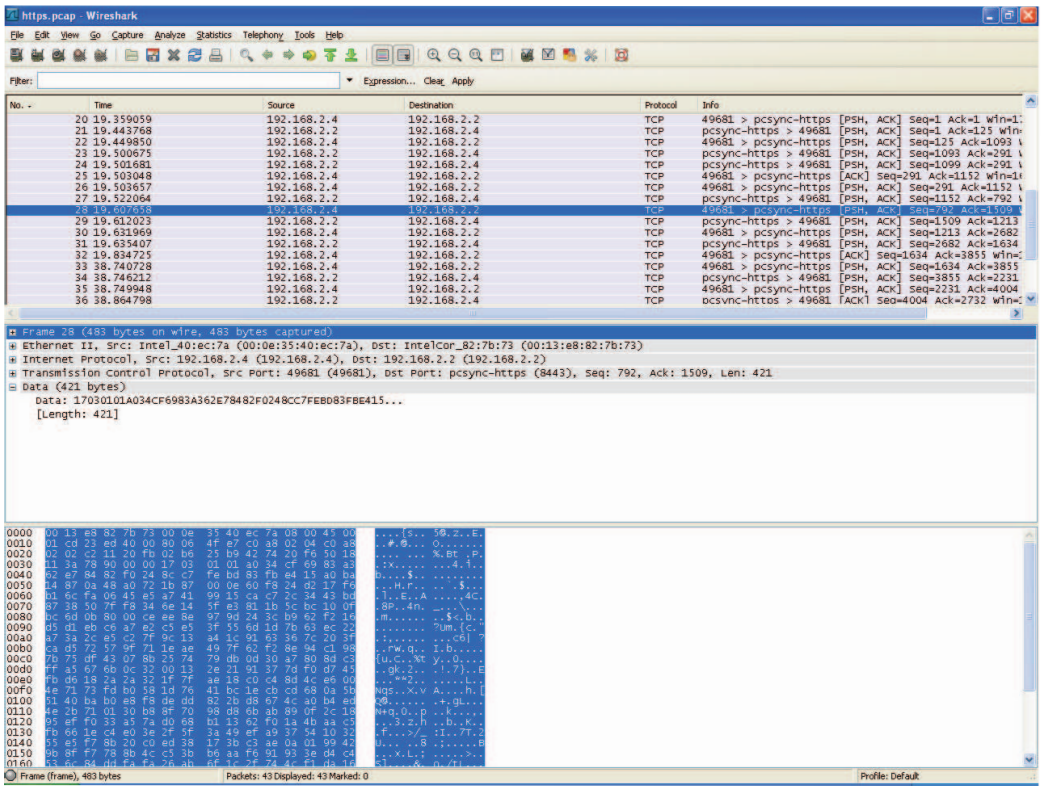
HTTP

Avlytting av kommunikasjon mellom klient- og serverapplikasjon som gikk over HTTP lot seg avlytte. Kommunikasjonen som ble fanget opp av Wirehark kunne leses i klartekst og omfattet blant annet brukernavn og

passord (se figur 5.5) som ble sendt som en del av en POST-forespørsel i forbindelse med innlogging.

HTTPS

Det ble også gjort forsøk på å avlytte kryptert kommunikasjon over HTTPS mellom server og klient. For å få til dette måtte det genereres et sertifikat som ble plassert på serveren. Ettersom sertifikatet ikke er signert av godkjente sertifiseringsmyndigheter, vil nettlesere som kommuniserer med serveren advare om dette. I Adobe Flash Players sikkerhetsmodell anbefales det ikke å hente data over HTTPS dersom Flex-applikasjonen/.swf-filen lastes ned over HTTP, men dette er allikevel teknisk mulig å få til dersom man spesifiserer crossdomain-filen korrekt (se 4.1.7 for forklaring på hva en crossdomain-fil er).



Figur 5.6: Avlytting av HTTPS med Wireshark

Resultatet av avlyttingsforsøkene viser at kommunikasjon over HTTPS ikke kan leses av i klartekst. Ettersom dataene var kryptert, lot det seg ikke

gjøre å lese noe meningsfylt ut av kommunikasjonen som ble fanget opp av Wireshark.

Oppsummering

Resultatene av avlyttingsforsøkene er som ventet. Vanlig HTTP er mulig å avlytte og er ikke anbefalt i tilfeller der datautvekslingen kan inneholde sensitiv informasjon. HTTPS-kommunikasjon kan ikke leses av i klartekst, og gjør det således mulig å i det minste sikre at selve kommunikasjonskanalen mellom klient og tjener er kryptert.

Kapittel 6

Oppsummering og videre arbeid

6.1 Oppsummering

Forsøkene i denne oppgaven hadde til hensikt å kartlegge temaet sikkerhet i rike internettapplikasjoner, samt å definere gode råd og fremgangsmåter («best practices») for utviklere som lager applikasjoner av denne typen.

Arbeidet ble fokusert rundt hvilke sikkerhetsutfordringer som har vært sentrale for tradisjonelle webapplikasjoner. Hensikten med dette var å undersøke om noen av disse sikkerhetssvakhetene også gjelder for RIAer. Det ble i arbeidet med oppgaven også utviklet to rike internettapplikasjoner for bruk i sikkerhetstestingen. De to applikasjonene har samme funksjonalitet, og ble utviklet ved hjelp av rammeverkene Microsoft Silverlight og Adobe Flex. I tillegg til sikkerhetstesting ble det gjort forsøk på å dekompilere de to demonstrasjonsapplikasjonene. Dekompileringen hadde til hensikt å demonstrere at klientsidekode ikke kan skjules for eventuelle angripere.

Det ble også utviklet en serversideapplikasjon i Java som kommuniserer med klientsideapplikasjonene ved hjelp av webservicer. Arkitekturen er delvis basert på REST-prinsipper. Webservicene sender data i XML-format.

Dekompilering av klientsidekode viste seg å være ukomplisert. Ved dekompilering av Adobe Flex-kode ble det benyttet et verktøy fra HP, SwfScan. Det ble også gjort dekompileringsforsøk med SourceTec Sothink SWF decompiler. Begge disse verktøyene var istand til å dekompilere applikasjonen. Det ble også forsøkt å dekompilere Microsoft Silverlight-applikasjonen, og også dette forsøket var vellykket og viste at applikasjonen var dekompilerbar. Selv om den dekompilerte koden ikke ble identisk

med den originale i alle tilfellene, viser forsøkene at man må ta utgangspunkt i at ingenting som står i klientsidekoden er hemmelig for en angriper.

Når det gjelder forsøkene med å teste demonstrasjonsapplikasjonene for tradisjonelle angrep og fallgruver, viste disse at det er fullt mulig å lage rike internettapplikasjoner som er sårbare for en rekke svakheter man allerede kjenner til fra tradisjonelle webapplikasjoner. En kunnskapsrik angriper vil ha mange måter å angripe en rik internettapplikasjon på, og er ikke begrenset til å måtte angripe via en klientsideapplikasjon skrevet i for eksempel Silverlight eller Flex. Dersom klientsideapplikasjonen kommuniserer med en serverside, vil det ikke være vanskelig for en angriper å omgå eventuelle sikkerhetsmekanismer i klientapplikasjonen og angripe serversiden direkte. Derfor er det viktig å være klar over at det først og fremst er på serversiden man kan implementere sikkerhetsmekanismer som gjør en applikasjon motstandsdyktig mot tradisjonelle angrep som for eksempel XSS (HTML-, XML eller JavaScript-injisering), SQL-injection og XSRF.

Å implementere sikkerhetsmekanismer som for eksempel inputvalidering i klientsidekoden kan være nyttig for å hjelpe brukerne med å bruke applikasjonen, men er ikke egnet til å utgjøre eneste sikkerhetsmekanisme. Årsaken til dette er at brukerne (og dermed også potensielle angripere) kan gjøre hva de vil på klientsiden. Det er først på serversiden at det er mulig å validere informasjon slik at den kan brukes videre.

Forsøkene viste at demonstrasjonsapplikasjonen som kjørte på serversiden var åpen for SQL-injection, XSRF og usikker direkte objektreferanse. Derfor var det mulig å foreta SQL-injection via både Flex- og Silverlightapplikasjonen. Det var i tillegg mulig å foreta XSS via Flex og XSRF direkte mot serverapplikasjonen. Forsøkene viste imidlertid at rammeverkene i liten grad led under informasjonslekkasjer eller viste omfattende tekniske feilmeldinger når feilsituasjoner oppsto.

Forsøk på å avlytte kommunikasjon over HTTP var vellykkede og viser at krypterte kommunikasjonskanaler bør benyttes ved utveksling av sensitiv informasjon. HTTPS var også mulig å avlytte, men dataene som ble avlyttet var krypterte og fremsto derfor som uforståelige.

6.2 Konklusjon

Etter å ha utviklet og sikkerhetstestet rike internettapplikasjoner er konklusjonen i denne oppgaven at rammeverkene for utvikling av denne typen webapplikasjoner i liten grad endrer landskapet for websikkerhet.

Som utvikler må man fortsatt implementere sikkerhetsmekanismer på serversiden dersom man ønsker å lage en sikker applikasjon. Klientsidevalidering kan være nyttig som et middel for å hjelpe brukerne til å benytte seg av webapplikasjoner på en effektiv måte, men egner seg ikke til å validere input, sett fra et serversideperspektiv. Rike internettapplikasjoner inneholder klientsidekode, og det ble i denne oppgaven funnet at slik kode er mulig og uproblematisk å dekompilere. Dekompilering vil i denne sammenhengen si at koden kan «gjenskapes» nært opptil til sin opprinnelige tilstand. Dette må utviklere som skriver klientsidekode være bevisst på, og på grunn av den høye tilgjengeligheten det viser seg at klientsidekode har vil det være svært uheldig å plassere sensitiv informasjon i denne koden.

6.3 Videre arbeid

Dersom det skulle arbeides videre med problemstillingen «Sikkerhet i rike internettapplikasjoner» hadde det vært ønskelig å se nærmere på hva bruken av forskjellige former for datautveksling mellom klient og server har å si for sikkerheten. Både Flex og Silverlight tilbyr flere ulike måter å utveksle data med serversiden på, og det ville vært en naturlig del av et videre arbeid å teste ut flere av disse.

Videre ville det vært naturlig å se nærmere på flere av de rammeverkene, teknologiene og teknikkene som finnes for RIA-utvikling, for eksempel Google Web Toolkit og JavaFX.

Dersom man hadde valgt å definere uttrykket «RIA» til også å omfatte skrivebordsapplikasjoner laget i for eksempel Adobe AIR eller Silverlight versjon 3, ville det være enda mer å ta tak i når det gjelder sikkerhet. Denne typen applikasjoner minner om RIAer som kjører i nettleseren, men de kan under visse forutsetninger fungere uten internettforbindelse. I tillegg kan de under visse forutsetninger ha rettigheter på lik linje med tradisjonelle skrivebordsapplikasjoner, som medfører at de for eksempel kan ha tilgang til det lokale filsystemet på brukerens datamaskin. Dette åpner for en rekke nye sikkerhetsrelaterte utfordringer.

Videre arbeid kunne også bestått av forsøk på å utvikle et rammeverk for utvikling av RIAer med sikkerhet i fokus. Ved utvikling av tradisjonelle webapplikasjoner har man tatt utgangspunkt i at serversiden må være forberedt på all mulig slags input fra klientsiden. Som vi har sett av resultatene fra sikkerhetsforsøkene i denne oppgaven endrer ikke rammeverkene for RIA-utvikling denne situasjonen. Derfor ville det være naturlig at serverapplikasjonen var hovedfokus i et eventuelt prosjekt av denne typen.

Dersom dekompilering av klientsideapplikasjonene ikke var mulig, ville sikkerhetssituasjonen for rike internettapplikasjoner se annerledes ut. Videre arbeid kunne derfor også bestå av å sette seg bedre inn i denne problemstillingen og undersøke nærmere om det finnes klientsidekode som ved hjelp av obfuskering eller andre teknikker gjør det mulig å forhindre dekompilering.

Kapittel 7

Sjekkliste for sikker RIA-utvikling

Dette kapittelet er en oppsummering av konklusjoner og råd fra utvikling og sikkerhetstesting av to rike internettapplikasjoner i rammeverkene Adobe Flex og Microsoft Silverlight. Vær oppmerksom på at andre rammeverk enn disse to kan ha andre svakheter og sikkerhetsmekanismer enn det som er forutsatt under utarbeidelsen av denne sjekklisten.

Dekompilering og obfuskering

Ønsker man å benytte Flex og Silverlight til utvikling av sikre, rike internettapplikasjoner, er det viktig å være klar over at koden man skriver kan dekompileres, det vil si kjøres igjennom et dekompileringsverktøy slik at opprinnelig programkode blir lesbar. Applikasjoner skrevet ved hjelp av disse rammeverkene kjøres i brukernes nettlesere på klientsiden og kan med enkle midler dekompileres, leses og endres av angripere. Derfor er det viktig å utvikle denne typen applikasjoner som om man skulle være nødt til å legge koden ut på internett i klartekst, ettersom det nesten er det man gjør når man publiserer en slik applikasjon.

Det har blitt gjort forsøk på å obfuskere applikasjonene, altså gjøre koden de inneholder uleselig for angripere. Disse forsøkene har ikke ført frem, og det anbefales ikke å stole på at slike verktøy kan gjøre klientsidekode uleselig for angripere.

En måte å begrense spredning av klientsidekoden på er å flytte autentiseringsmekanismen fra den rike internettapplikasjonen til en vanlig webside. Klientsideapplikasjonen kan sendes til brukerens nettleser som respons på en vellykket innlogging. På den måten unngår man at en eventuell angriper får tilgang til klientsideapplikasjonen før hun/han er au-

tentisert. En slik løsning forhindrer ikke at autentiserte brukere får tilgang til klientsidekoden, men kan begrense spredning av for eksempel en bedriftsintern RIA.

Cross-Site Scripting (XSS)

Cross-Site Scripting gjør det mulig for angripere å inkludere ondsinnede script i en webapplikasjon. Microsoft Silverlight er ikke utsatt for XSS med mindre man bevisst lar brukeren inkludere JavaScript i HTML-siden som omgir Silverlight-applikasjonen. Flex kan være sårbar for XSS ved bruk av metoden «navigateToUrl», men bare dersom brukeren aktivt klikker på linken som fører til XSS-angrepet. Løsningen er å validere inputparametre til denne metoden. Ved bruk av Flex-rammeverket bør man benytte versjon 3.4 eller nyere grunnet en XSS-feil i selve rammeverket i tidligere versjoner.

SQL-injection

SQL-injection er en angrepstype hvor angripere ved hjelp av spesiallaget input kan endre databasespørringer og i verste fall få full tilgang til databasen. Angrep unngås ved å «escape»/uskadeliggjøre alle kontrolltegn. For å være sikker på hvilke tegn dette gjelder, sjekk hvilke kontrolltegn det aktuelle databasesystemet benytter. Et godt alternativ til manuell håndtering av kontrolltegn er å benytte «prepared statements», men husk isåfall at kombinasjonen jokertegn og bruk av «LIKE»-sammenlikningsoperatoren igjen gjør applikasjonen sårbar for SQL-injection.

Usikre direkte objektreferanser

Dette er en fallgruve som gjør det mulig for angripere å få urettmessig tilgang til data. I de tilfellene hvor det hentes data fra databasen basert på informasjon som har vært på klientsiden, må det sjekkes om den som forespør dataene egentlig skal ha tilgang til dem. For eksempel må en webservice som henter data ut i fra en identifikator som sendes fra klientsiden, sørge for å kontrollere at den som sender forespørselen faktisk skal ha tilgang til de dataene som etterspørres.

Cross-Site Request Forgery

XSRF går ut på at en angriper skreddersyr en forespørsel og forsøker å lure et offer til å sende den til en sårbar webapplikasjon. Er applikasjonen åpen for XSRF, kan angriperen sende forespørsler via offeret sitt, som må være innlogget for at angrepet skal fungere. Verken Microsoft Silverlight og Adobe Flex ser ut til å være sårbare for XSRF. Det kan derimot en eventuell serverapplikasjonen være. Det er derfor nødvendig å implementere et «billettsystem» for alle POST-forespørsler. Billettsystemet fungerer ved at det generérer en vilkårlig verdi som sendes til klientapplikasjonen, og som denne sender tilbake igjen som en del av en eventuell, påfølgende POST-forespørsel til serversiden. Slik kan man sikre at forespørselen kommer fra klientapplikasjonen og dermed unngår man XSRF.

Utilstrekkelig feilhåndtering og informasjonslekkasjer

Utilstrekkelig feilhåndtering kan føre til feilmeldinger som avslører teknisk informasjon. RIA-rammeverk er ikke spesielt utsatt for informasjonslekkaser, men husk å håndtere alle feilsituasjoner uten å avsløre teknisk informasjon en sluttbruker ikke vil ha glede av. Adobe Flex håndterer en rekke feil automatisk. Ved utvikling av Silverlight-applikasjoner må kode som foretar feilhåndtering skrives manuelt.

Utilstrekkelig autentisering og sesjonshåndtering

Autentiseringsmekanismen i en applikasjon vil ikke være sterkere enn det svakeste leddet. Unngå «Glemt passord»-funksjonalitet som gjør det enkelt for en angriper å stjele eller gjette passordet. Husk å avslutte eller ugyldiggjøre sesjonsobjektet når brukeren logger ut. Det kan også være en god idé å kontrollere og eventuelt redusere tiden det tar før en inaktiv sesjon blir avsluttet automatisk.

Avlytting av datatrafikk

Kommunikasjon over HTTP er mulig for angripere å avlytte. Benytt derfor HTTPS når klient og server utveksler sensitiv informasjon.

Kapittel 8

Vedlegg

8.1 Obfuskering

8.1.1 Ikke obfuskert kode

Følgende kode er ikke obfuskert før dekompilering:

```
private function _WishList_HTTPService6_i() : mx.rpc.http.mxml::HTTPService
{
        var loc0:* = new HTTPService();
        this.register_user = loc0;
    loc0.method = "POST";
    loc0.url = "http://localhost:8080/WishList/register";
    loc0.useProxy = false;
    loc0.request = 0;
    loc0.addEventListener("result", this.__register_user_result);
    loc0.addEventListener("fault", this.__register_user_fault);
    loc0.initialized(this, "register_user");
    return loc0;
}
```

8.1.2 Amayeta SWF Encrypt 6.0

Følgende kode er obfuskert ved hjelp av Amayeta SWF Encrypt før dekompilering:

```
private function _WishList_HTTPService6_i() : mx.rpc.http.mxml::HTTPService
{
    var loc0:* = new HTTPService();
    this.register_user = loc0;
    loc0.method = "POST";
    loc0.url = "http://localhost:8080/WishList/register";
    loc0.useProxy = false;
    loc0.request = 0;
    loc0.addEventListener("result", this.__register_user_result);
    loc0.addEventListener("fault", this.__register_user_fault);
    loc0.initialized(this, "register_user");
    return loc0;
```

```
}
```

8.1.3 Ambiera Irrfuscator 2.0

Følgende kode er obfuskert ved hjelp av Ambiera Irrfuscator før dekompilering:

```
private function _br10459() : mx.rpc.http.mxml::HTTPService
{
    var loc0:* = new HTTPService();
    this._wu7793 = loc0;
    loc0.method = "POST";
    loc0.url = "http://localhost:8080/WishList/register";
    loc0.useProxy = false;
    loc0.request = 0;
    loc0.addEventListener("result", this._tr4156);
    loc0.addEventListener("fault", this._nw10508);
    loc0.initialized(this, "_wu7793");
    return loc0;
}
```

Bibliografi

[1] Adobe. Adobe player statistics. http://www.adobe.com/products/player_census/flashplayer/version_penetration.html
Besøkt 15/05/2009.

[2] Jeremy Allaire. Macromedia flash mx - a next generation rich client. 2002.

[3] Amayeta. Swf encrypt 6.0. www.amayeta.com.

[4] Ambiera. irrfuscator actionscript 3 obfuscator. www.ambiera.com.

[5] Adam Bixby. Adobe flex 3.3 sdk dom-based xss:
http://www.gdssecurity.com/l/b/2009/08/20/adobe-flex-3-3-sdk-dom-based-xss/
besøkt 12/10/2009.

[6] Caplex. Integritet. 2000.

[7] Alaric Cole. *Learning Flex 3: Getting up to Speed with Rich Internet Applications*. O'Reilly, 2008.

[8] David Gassner. *Flex 3 Bible*. Wiley Publishing Inc., 2008.

[9] Rickland Hollar og Richard Murphy. *Enterprise Web services security*. Charles River Media, Hingham, MA, USA, 2005.

[10] RIAstats.com (http://www.riastats.com). Besøkt 20/09/2009.

[11] Sverre H. Huseby. *Innocent code: a security wake-up call for Web programmers*. John Wiley and Sons, Inc.

[12] Adobe Systems Inc. Security bulletin:
http://www.adobe.com/support/security/bulletins/apsb09-13.html
besøkt 12/10/2009.

[13] www.digi.no/Marius Jørgenrud. 316 norske nettsteder hacket siste måned. World Wide Web electronic publication, 2008.

[14] James F. Kurose og Keith W. Ross. *Computer Networking: A top-down approach*. Pearson/Addison Wesley, 2008.

[15] Microsoft. Microsoft releases silverlight 2. http://www.microsoft.com/presspass/press/2008/oct08/10-13silverlight2pr.mspx
besøkt 20/09/2009.

[16] Microsoft. Microsoft silverlight faq. http://www.microsoft.com/silverlight/resources/faq.
besøkt 25/09/2009.

[17] BBC News. Hacket partikkelakselleratoren på cern. World Wide Web electronic publication, 2007.

[18] Rain Forest Puppy. Nt web technology vulnerabilities. *Phrack Magazine*, 1998.

[19] www.aftenposten.no/Roald Ramsdal. 60 000 personnumre på avveie. World Wide Web electronic publication, 2007.

[20] Billy Kim Rios og Raghav Dube. Kicking down the cross domain door. techniques for cross domain exploitation. Mars 2007.

[21] Joel Scambray, Mike Shema og Caleb Sima. *Hacking exposed: Web applications*. McGraw-Hill.

[22] Andrew van der Stock. Owasp list of top 10 web application vulnerabilities for 2007. World Wide Web electronic publication, 2007.

[23] John Stone og Sarah Merrion. Instant messaging or instant headache? *Queue*, 2(2):72–80, 2004.

[24] Art Taylor, Brian Buege og Randy Layman. *Hacking exposed: J2EE & Java*. McGraw-Hill, 2002.